Updating Compressed Column-Stores

Heman, S.A.B.C.

2015

document version
Publisher's PDF, also known as Version of record

Updating Compressed Column-Stores

Sándor Héman

Committee: prof.dr. Herbert Bos
dr. Götz Graefe
prof.dr. Martin Kersten
prof.dr. Thomas Neumann
dr. Jacopo Urbani

The research reported in this thesis has been partially carried out at *CWI*, the Dutch National Research Laboratory for Mathematics and Computer Science, within the theme *Database Architectures*.

The research reported in this thesis has been partially carried out as part of the continuous research and development of the Vectorwise database management system from Actian Corp.

SIKS Dissertation Series No.2015-27
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The research reported in this thesis was partially funded by NWO.

The cover was designed by Robin Héman, OKOKdesign.

The printing and the binding of this disseration was carried out by CPI Wöhrmann print service.

ISBN 978-94-6203-876-9

VRIJE UNIVERSITEIT

Updating Compressed Column-Stores

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. F.A. van der Duyn Schouten,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op woensdag 28 oktober 2015 om 15:45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

Sándor Antal Bernardus Christophorus Héman

geboren te Amsterdam

promotor: prof.dr. P.A. Boncz

To my parents, Eva and Ben,
my wife, Iris,
and our newborn son, Oscar.

Contents

Acknowledgments

During my computer science studies at university, my main interests were in the areas of hardware architecture, systems and algorithms/data structures. Inspired by the lectures of Martin Kersten (thank you for those!), I figured that the field of database systems would satisfy all my interests, so I applied at his research group within CWI. This is where I got introduced to Peter Boncz, initially as my copromotor. It quickly became clear to me that Peter's brain operates at stunning speeds, just like the database kernel (MonetDB/X100) he was researching. He thought me the ins and outs of hardware-conscious algorithm design through a torrent of "crazy" and unconventional, but at the same time clever and inspiring, ideas. The first one that caught my interest, was the use of data compression to help keep his extremely "data hungry" X100 database engine from starving. With a hint that I should forget about theoretically optimal constructs, like Huffman codes, and rather focus on raw CPU cycles and experimentation.

From then on, I was part of the small, hard-working team that did research in the area of high-performance analytic database architecture. Together with Marcin Żukowski, Niels Nes and Peter himself, we spent many days (and nights!) brainstorming, hacking, benchmarking and writing papers. Through Arjen de Vries and Roberto Cornacchia, we also had several "flirts" with the field of information retrieval. I would like to thank the ones involved for these exciting and challenging moments.

Eventually, however, after focusing on all kinds of data- and compute-intensive optimizations, I was starting to feel a bit of a tension. Yes, we had a fast engine. And, yes, we had a heavily scan optimized storage layout. However, we were silently ignoring "That-What-Must-Not-Be-Named": column-store updates. Still young and adventurous, I decided to take up the challenge, and attack the mysteries around transactional column-store update support. Soon after this "defining" moment in my PhD career, things took a bit of a turn, however.

The year 2008 was the year of both an internship at HP Labs and the Vectorwise spin-off. First things first, however, as 2008 was also the year I got married to my beloved wife, Iris. Thank you Iris for all the love, support and care, during both the good and the darker times. Our wonderful honeymoon in California was followed directly by a research internship at HP Labs. I would like to thank Götz Graefe, Umesh Dayal and Meichun Hsu for providing me this opportunity in the first place, and Goetz in particular, for sharpening my knowledge in the areas of B^+-trees and fault tolerance. Thanks also to Dimitris, Stavros and Mehul for showing me around "the Valley".

After returning from the US, the Vectorwise spin-off company was already up and running. This meant that, besides getting the work on column-store updates published, it now had to be made production ready as well. Luckily, we were now in a position to hire more people, and I would like to thank Hui Li and Michał Szafranski for helping out with checkpointing, Gosia Wrzesinska for help around transaction management and Lefteris Sidirourgos for his help with publishing a paper during these busy times. An apology is in place towards Gosia, for the nightmares we both had around "join index updates" ;-) Thanks also to Giel de Nijs, not only as a colleague, but also as a "biking buddy". The

early morning and evening rides provided a welcome break from office and urban life. For getting Vectorwise where it stands today, I would furthermore like to thank: Adrian, Ala, Andrei, Bill, Bogdan, Christian, Christophe, Dan, Dick, Doug, Emma, Fred, Ian, Irish, Julek, Kamil, Karl, Michal, Roger, Tim, Willem, Zelaine, and all the others I did not meet in person.

After several years at Vectorwise, unfortunately, I had to lay low for some time due to a prolonged period of illness. When feeling sufficiently recovered, I decided it was "now or never", if I ever wanted to finish this thesis. Although a significant amount of relevant development and engineering work was conducted during my time at Vectorwise, the writing part got mostly neglected. I would like to thank both Peter Boncz and Stefan Manegold for taking me back into the Database Architectures group at CWI, which allowed me to finish this work in relative peace and quiet, and to finally connect all the pieces of the puzzle around "join index updates". Furthermore, I would like to thank all fellow students and researchers, both past and present, from the Database Architectures group at CWI. It has been a great place to work, full of smart people that were a joy to be around, both during and outside work hours.

During my, admittedly long, journey of almost 10 years, Peter received the recognition he deserves for his hard work in both academia and industry, recently obtaining the status of full professor. The consequence being, that Peter can now be my promotor. It is an honor to me to (hopefully!) become the first to receive a doctorate degree from him.

Hugs and kisses go to all my family members, without whom I would have never been able to complete this lengthy educational career. Thanks for all the support, during both the good and the hard times, to my mother and father, Eva and Ben, brother and sister, Robin and Giulia, my wife, Iris, and my parents- and sister in law, Liesbeth, Arnold and Lucia.

Finally, I would like to thank all committee members, Herbert Bos, Götz Graefe, Martin Kersten, Thomas Neumann and Jacopo Urbani for the time and effort spent to take part in this defense.

Chapter 1

Introduction

Since the advent of our modern information society, gathering and organizing digital information has been an important task. Over the last few decades, an increasing amount of information and activity has been digitized. Examples include the management of customer accounts at a bank, processing of e-commerce transactions by a web shop, collection of geographic and other sensor data from our surroundings, and human interaction through social networks. This has resulted in an exponential growth in the volume of data that needs to be managed by companies and organizations.

Luckily, this data explosion has been accompanied by gradual advances in computer hardware. The capacities of primary and secondary storage devices have grown along with data volumes exponentially, as has the computational power of general purpose processors. However, the development of data transfer speeds between the layers of a hierarchical memory system has been lagging behind. For (random) access latency, the situation is even worse, showing only marginal improvements over the years. The result is a widening gap between raw computational power and the rate at which data can be retrieved from storage. For large bodies of data, typically stored on secondary storage, this is a growing problem. Therefore, satisfying the general desire from both end-users and owners of information systems to have fast access to rapidly increasing amounts of data, is becoming more and more of a challenge, to be solved through clever software design.

1.1 Database Management Systems

Since the 1970s, *database management systems* (DBMS) have been used to organize data on a computer system in a structured fashion, providing standardized mechanisms to query and manipulate the underlying data. The prevalent model to represent information in a DBMS has been the *relational model* [Cod70], which organizes data into *relations*, containing *tuples* built from a set of related *attributes*. For example, to represent information about customers of an organization, a *customer* relation might hold tuples with five attributes of the form *(customer-id, name, street, zip-code, city)*. In a DBMS, a relation is typically modeled as a two-dimensional table, where each *row*, or "record", represents a single tuple, and each *column* represents a single attribute.

1.1.1 On-line Transaction Processing (OLTP)

The long history of relational technology has left behind a legacy of systems that are geared towards *on-line transaction processing* (OLTP), one of the early application domains for DBMSs. OLTP focuses on efficient retrieval and manipulation of a limited amount of records within the database, for example to implement a banking transaction that transfers funds from one account to another. To deal with such tuple level operations conveniently, traditional DBMSs employ a record-oriented storage model, where attribute values of a single tuple are stored contiguously within a disk page (the minimum fixed-size unit for data transfers to and from secondary storage). Auxiliary disk-resident indexing structures are used to speed up access to the pages containing records of interest. The result is a write-optimized "row store" architecture, where each record can be easily updated in-place, using a single write to disk. Today, this is still the prevalent architecture for systems geared towards OLTP, where sustaining a high throughput on workloads consisting of fine-grained transactional changes is the top priority.

1.1.2 Analytic Query Processing

Since the 1990s, a desire to gain more high-level insight into the data gathered in a DBMS has led to the advent of application domains that are more analytic in nature. Under the common umbrella of "business intelligence" (BI), fields like *on-line analytic processing* (OLAP), reporting, analytics and data mining share a common goal of extracting meaningful information from large amounts of data, to provide, for example, historical, current, or predictive views of business operations. In the academic world, similar interests can be found within the field of "data science", where research domains like astronomy, biology, health care and social sciences all use information technology to extract knowledge from gathered data. From a database perspective, this calls for an ad-hoc, exploratory and "read-mostly" type of query processing, often over large, i.e. gigabytes to petabytes, volumes of data.

Until the early 2000s, "one size fits all" used to be the leading philosophy among DBMS vendors. This resulted in separate instances of a traditional DBMS being used independently of each other, an on-line one for day-to-day business, and an off-line *data warehouse* for analytics. The idea being that heavy analytic queries should not interfere with traditional business transaction processing, and vice versa. One problem with this approach, is that the analytic DBMS typically operates on outdated tables, as the data warehouse is not being updated in real-time, but rather only "refreshed" by batches of updates at chosen intervals, for example outside office hours.

A second problem lies in the use of traditional row store DBMSs for analytics. Analytic queries tend to be compute- and data-intensive, typically involving the computation of aggregates over many tuples, but only a subset of attributes. A traditional record-oriented page layout therefore wastes precious I/O bandwidth by always reading *all* attribute values from storage, rather than only the ones of interest to a query. This results in suboptimal performance on data-intensive workloads. Furthermore, research has shown that the traditional systems perform poorly in terms of CPU utilization [ADHW99, PMAJ01], making them score badly on compute-intensive workloads as well.

For those reasons, the "one size fits all" philosophy has come to an end [SC05], with DBMSs that use columnar storage (DSM [CK85]), or *column stores*, holding the future for performance critical analytics [SR13].

1.1.3 Column Stores

Over the last decade, column-store DBMSs have regained significant interest, with C-store [Sto05] and Sybase IQ [Syb] being two well known examples from research and industry, respectively. C-store eventually got commercialized as Vertica [Ver], while other systems, like IBM's DB2 with BLU acceleration [RAB+13], Microsoft's SQL Server [LCF+13], Infobright [SE09], Paraccel [Par], and Exasol [Exa] entered the column-store market. The MonetDB/X100 [BZN05] research column-store, and its commercial descendant, Vectorwise [ZvdWB12], provide the context for this thesis, and are introduced in the next section.

In a column-oriented DBMS, attribute values from a *single column* are stored consecutively within a page on disk, filling as many pages as needed to store all values that make up the column. The columns of a table are therefore represented as disjoint sets of column pages. The main advantage of this layout is that scans involving only a subset of attributes can be restricted to read relevant column pages only, potentially leading to a significant reduction in scanned data volume compared to traditional row-stores. Furthermore, pages with attribute values lend themselves well to data *compression*, as all values are of equal data type and originate from the same domain. Compressibility and scan locality can often be improved even further by keeping tables *sorted* in one or more interesting sort orders.

The resulting table layout is well suited for scan- and data-intensive read-only workloads, but notoriously unfriendly towards *in-place updates*. For example, insertion of a new tuple into a table with N attributes would require N writes to disk, as each attribute value goes to a different page. In a row-store system, for comparison, a single write suffices, irrespective the number of attributes. In case of compressed and/or sorted tables, in-place column-store updates become even more problematic, as pages would have to be decompressed, updated, and recompressed, while sort orders might call for expensive shifting of tuples.

Therefore, most column-store systems employ a differential approach to updates [SL76], where delta changes with respect to an immutable "master" image of a table are maintained separately, like the errata to a book. In general, this results in an architecture with a column-oriented *read-store*, and an update-friendly *write-store* [HLAM06], which is often implemented as an (indexed) row-store. During a scan, updates from the write-store can then be *merged* into a scan of the read-store to produce an up-to-date table image.

1.1.4 MonetDB/X100 and Vectorwise

The research described in this thesis takes place within the context of MonetDB/X100 [BZN05], a prototype column-store system from the database group of CWI, the Dutch research institute for mathematics and computer science. MonetDB/X100 was designed from scratch for high performance analytics on modern hardware. Besides scientific publications, work on MonetDB/X100 has

resulted in the Ph.D. thesis of Marcin Żukowski [Żuk09] and a spin-off company called *Vectorwise*. Vectorwise was acquired by Ingres (now Actian) in 2011 [IZB11], where research, development and sales of the system continue. For the remainder of this thesis, we stick with the name Vectorwise, even when referring to the pre-acquisition MonetDB/X100 research prototype.

Vectorwise has its roots in MonetDB [Bon02], one of the pioneers within the field of column-oriented database systems, which is still being actively maintained and available for download under an open-source license[1]. MonetDB differs from "conventional" column-stores in its focus on main-memory query execution rather than the potential I/O benefits discussed in the previous section. Its column-oriented memory layout allows an attribute to be represented by a contiguous memory region, or "array", which can be computed upon using compiler-friendly and CPU-efficient loops. This *column-at-a-time* execution model makes heavy use of sequential memory access patterns, which is beneficial in terms of transfer bandwidth when *reading* cache-lines from memory into the CPU, due to the combination of spatial and temporal locality of access. The main drawback, however, is that intermediate results are also *written* back to memory column-at-a-time, only to be read back into the CPU on subsequent computational steps. Therefore, this execution model is still susceptible to the danger of memory bandwidth becoming a bottleneck.

Vectorwise improves upon MonetDB by employing a *vectorized* execution model, where computational loops restrict themselves to fragments, or *vectors*, of columnar data, both as input and output. By picking a vector size that allows vectors to fit the hardware cache(s) of a CPU, materialization of intermediate results to main memory can be avoided, while still retaining the benefits of a sequential memory access pattern. Furthermore, Vectorwise adds the ability to scale beyond main-memory, by employing a column-oriented buffer manager, responsible for arbitrating access to database tables of "unlimited" size. This way, Vectorwise aims to benefit from both the I/O and CPU advantages associated with a columnar table layout. And not without success, as illustrated by entering the official TPC-H rankings[2] in top position on the 100GB, 300GB, and 1000GB benchmarks (including innovations presented in this thesis).

1.2 Research Questions

A vectorized, column-oriented query engine, like Vectorwise, is capable of exposing large amounts of computational power, which makes the engine extremely "data hungry". Given the growing discrepancy between the computational power of CPUs and available I/O bandwidth from secondary storage, delivery of data to the CPU can be challenging, even in a scan-optimized column-store. The first research question addressed by this thesis is therefore:

Question 1: Can light-weight data compression be used to alleviate I/O bottlenecks in a column-oriented DBMS?

Storing attribute values in a compressed format makes column-stores even more update-unfriendly. The second research question therefore becomes:

[1] www.monetdb.org

[2] www.tpc.org/tpch/results/tpch_perf_results.asp

Question 2: How can we add real-time, transactional update support to a compressed column-store, in such a way that any negative impact on the performance of read-only queries is minimized?

1.3 Thesis Outline and Contributions

This thesis is organized in three parts: background (Chapters 2- 4), column compression (Chapter 5) and column-store updates (Chapters 6-7).

Background and introduction to Vectorwise. Chapter 2 provides an overview of concepts and trends in computer hardware architecture that are relevant to later chapters, and the design philosophy of Vectorwise in general. A recap of relational database theory and execution engine architecture is given in Chapter 3. Chapter 4 then describes the design of Vectorwise, the DBMS used for experimental evaluation in the remainder of the chapters. Here, we combine knowledge from the first two background chapters to motivate and describe the *vectorized in-cache execution model* [BZN05] of the Vectorwise engine. Besides, this chapter describes the storage layout, buffer management policies and indexing mechanisms specific to Vectorwise.

Light-weight column compression.[3]. Chapter 5 addresses research question 1, and was published in ICDE'06 [ZHNB06].
Here we present several *light-weight column compression* methods, together with on-demand *into-cache decompression*, that can be used to improve execution times of I/O bound queries. The idea is to read compressed pages from disk, and decompress them with minimal CPU overhead, so that perceived I/O bandwidth can be improved. To minimize CPU overhead of decompression, we observe that pages in a column-store contain data from the same domain and of a single, known data type. We use this knowledge to design simple, but very fast, compression schemes, capable of decompressing gigabytes of raw data per second. Furthermore, to avoid materialization of entire decompressed pages in main memory, we decompress each page in small fragments, i.e. "vectors", that fit the CPU cache, in an on-demand fashion, only when data is consumed.

Positional differential updates for column-stores. Chapter 6 addresses research question 2, and was published at SIGMOD'10 [HZN+10] and patented by Actian Corp [HBZN10, HBZN13b, HBZN13a].
To add transactional update support to a column-store DBMS, we consider in-place updates to be out of question, due to I/O costs that are proportional to the number of table attributes. Rather, we focus on differential update techniques [SL76], where delta updates against the immutable tables in a scan-optimized read-store are maintained in an update-friendly write-store. During a scan, write-store updates are merged with the base read-store data to produce an up-to-date table image.

We propose to organize delta updates against a read-store table by update *position*, or tuple offset, in a novel index structure called *positional delta tree* (PDT). Alternatively, one could organize update tuples by their (sort) key attribute *values*, for example in a B+-tree-like structure. We show that positional

[3] Joint work with Marcin Żukowski, also appeared in his PhD thesis [Żuk09]

update maintenance has several advantages over a value-based approach, most notably in the area of merging efficiency during a table scan. Furthermore, PDTs allow efficient encoding of attribute level modifications (i.e. SQL UPDATE), and can be stacked on top of each other, which allows for a clean implementation of snapshot isolation to support transactions.

Finally, Chapter 7 addresses the topic of index maintenance. Most notably, we contribute a compressed, positional representation of the original join index [Val87], which can be maintained efficiently under updates. Furthermore, we provide solutions to several subtle concurrency issues that surfaced during the integration of PDTs into the Vectorwise system.

Chapter 2

Hardware overview

This chapter is intended as a quick review of computer hardware architecture. It only touches upon technologies briefly, and is aimed at making the reader familiar with the concepts needed to understand the research presented in this thesis. The text focuses on standalone desktop/server-style computer systems. Networked, embedded or mobile devices are considered out of scope. For a more in-depth introduction to computer architecture, the reader is referred to textbooks like [HP11, HP12].

2.1 Introduction

When using the term computer, we usually do not refer to a single, clearly defined object, but rather to a collection of hardware components resembling those depicted in Figure 2.1. The most important components are the *processor*, *main memory*, and *disk*. The remaining components are *input/output* (I/O) devices that are intended to communicate with the outside world, making the computer a more useful device that is able to receive input from users and display graphical output on a screen. A *logic board* is typically used to connect all the components together, and provides the necessary communication infrastructure like main-memory and I/O buses and controllers.

This hardware is managed by a collection of software called an *operating system*, or OS [Sta09]. The OS is responsible for controlling hardware resources, handling I/O requests, and providing services to user programs and scheduling those. A running (i.e. "active") program, often called a *process*, can issue a request for an OS service by means of a *system call*.

The processor, or more formally *central processing unit* (CPU), is responsible for execution of the stream of instructions that make up a computer program. CPUs are discussed in detail in Section 2.4. Main memory, also called *primary storage*, *random access memory* (RAM), or simply *memory*, holds both running programs and the data these operate on during their execution. Memory is the topic of Section 2.5. Primary storage can only hold data while the computer is running, i.e. it needs electrical current to maintain its state and is therefore volatile. To store programs and data permanently, a second layer of storage is introduced: *secondary storage*. This layer includes several *hard drive* or "disk" technologies that are outlined in Section 2.6.

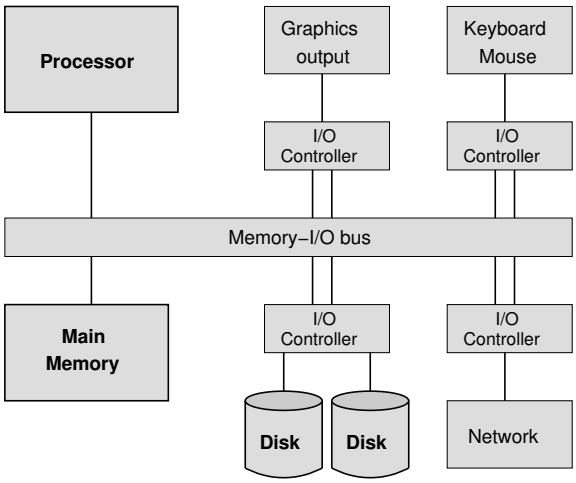


Figure 2.1: General purpose computer with connected I/O devices.

2.2 Latency and Bandwidth

Two important concepts when evaluating the performance of computer hardware are *latency* and *bandwidth*.

latency is a measure of time. It is defined as the amount of time a certain system component has to wait for the response to a certain request. Such waiting time is wasted time, and is therefore undesirable.

bandwidth is a measure of *throughput*. In computing it typically denotes the bit-rate at which two system components can communicate. It can be expressed in either bits or bytes per second, or multiples thereof.

These definitions are not hard, as both terms tend to be used more freely, making their interpretation context dependent. For example, in the context of a CPU, latency can refer to the time that is needed to execute a single instruction, and bandwidth, or throughput, would be used to indicate the number of instructions being completed per second. When dealing with disk drives, however, latency refers to the time needed to locate and retrieve a certain piece of data on the hard drive. Bandwidth and throughput then refer to the amount of data per second that the drive is capable of providing to the consumer.

2.3 Data Representation

A computer stores data in binary representation, as a sequence of *bits*, which can hold a value of either zero or one. Handling of bits can be implemented conveniently in digital electronic circuitry through logic gates that use, for example, a low and high voltage level to represent zero and one respectively.

Using bits, any number, and therefore any encodable piece of data, can be represented as a *binary number*, a number in base two. An n-bit binary number represents $\sum_{i=0}^{n} b_i 2^i$, where b_i is the value of the i'th bit, counting from right to left, with the least significant bit on the right. For example, the 8-bit number

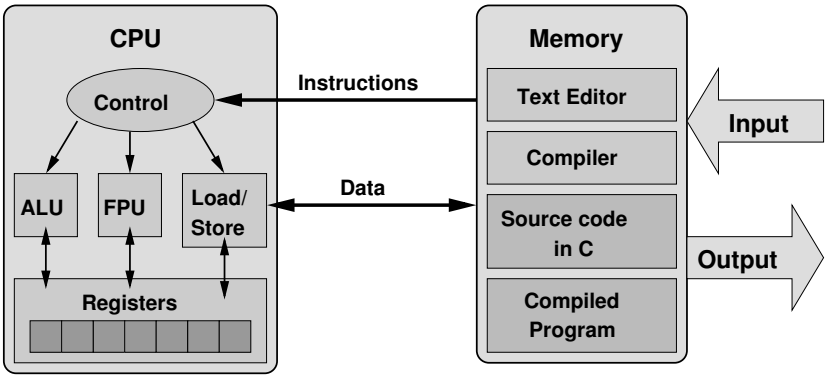


Figure 2.2: Schematic of stored program computer.

`10010110` represents $1\times2^7+0\times2^6+0\times2^5+1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0$, which equals 150 in decimal.

The smallest addressable unit of data in a computers memory is typically a *byte* (B), which equals eight bits. Bytes are used as building blocks for larger *data types*, for example four bytes for a 32-bit floating point number, eight bytes to hold a 64-bit integer, or an arbitrary number of bytes to represent objects of variable size. Each processor has a "natural" unit of data that it uses for its computations, which is called a *word*. I.e. 32- and 64-bit processors use a word size of 32 and 64 bits respectively. The word size of a CPU typically equals the maximum size of a memory address, which, in case the minimal addressable unit is a byte, limits the maximum size of addressable memory in bytes. For example, the maximum 32-bit number is $2^{32}-1 = 4294967295$, which means that a 32-bit machine can not address more than that number of bytes.

For large multiples of bytes we use the *Kilobyte* (KB) to mean $2^{10} = 1024$ bytes. Along the same lines, a *Megabyte* (MB) is $2^{20} = 1024^2 = 1024KB$, a *Gigabyte* (GB) is $2^{30} = 1024^3 = 1024MB$, a *Terabyte* (TB) which is $2^{40} = 1024^4 = 1024GB$ and a *Petabyte* (PB) is $2^{50} = 1024^5 = 1024TB$.

2.4 CPU Architecture

A central processing unit (CPU) is the core computational unit in a computer. It consists of an integrated circuit (IC) on a piece of semiconductor material, typically silicon, which is also known as a *"chip"*. A chip contains many *transistors*, which are used to switch electronic currents on or off, allowing implementation of more complex components like an *arithmetic and logic unit* (ALU) for integer arithmetic or a *floating point unit* (FPU) for floating point arithmetic.

Figure 2.2 shows a simplified model of a CPU that implements the popular *stored program computer*. In this model both a *computer program* and the *data* it operates on are stored outside the CPU, in main memory. A program consists of a stream of *instructions* that process the programs data. These instructions are decoded and executed by the CPU, thereby controlling the operations it performs.

2.4.1 Instructions and Clock Cycles

The instructions that a CPU understands are defined by its *instruction set architecture* (ISA). A programmer is responsible for writing such programs, either directly in a textual representation of machine code instructions, called *assembly language*, or by having a *compiler* translate a program from a higher level programming language, like C or C++, into machine readable instructions.

Execution of a program boils down to a sequential walk through the instructions that make up the program, executing each instruction that comes along, in lock-step with the CPUs *hardware clock*, which runs at some constant rate. The interval between two subsequent ticks of this clock is called a *clock cycle*.

Often, an instruction will modify some piece of *data*. In most modern architectures this can only be done if the data resides in special memory locations inside the CPU, called *registers*. For example, in the MIPS instruction set [MIP], we can add the contents of two source operand registers, `R1` and `R2`, writing the result into destination register `R3` using an instruction of the form:

```
ADD R3, R1, R2
```

We can broadly categorize instructions into three classes:

load/store instructions are responsible for moving data back and forth between RAM and CPU registers.

arithmetic instructions perform mathematical or logical operations on data loaded into registers.

branch instructions influence the next instruction to be executed by introducing jumps in the flow of instructions.

Two major differing instruction set architecture paradigms exist, RISC and CISC. *Reduced instruction set computing* (RISC) adheres to a philosophy where small instructions that perform simple tasks are considered desirable, as these can be executed at higher speeds. Also, RISC systems employ a load/store architecture, where data needs to be explicitly loaded into registers, by means of designated instructions, before it can be used by arithmetic and logic instructions. The MIPS [MIP] instruction set used throughout the code listings is a popular example of a RISC instruction set.

The opposite of RISC is *complex instruction set computing* CISC. In this paradigm, a single instruction can execute multiple low-level operations. For example, loading a piece of data from memory, performing arithmetic on it, and writing back the result can be expressed in a single instruction. Also, CISC supports more complex memory addressing modes. The most common desktop and server instruction set of today, Intel's *x86* [Int13], belongs to the CISC family. Both Intel and AMD processors implement this instruction set. Although modern x86 CPUs still expose the traditional CISC interface to the outside world for backwards compatibility, it is common for actual implementations to split complex CISC instructions into RISC-like *micro-ops* during execution.

In its simplest form, a CPU can execute one instruction per *clock cycle*. Such a design is not built for performance, however, as all logic has to be performed in a single clock cycle, which limits the CPUs clock rate. A couple of architectural changes were introduced to boost processor performance. This was mainly done by allowing for more parallelism during program execution.

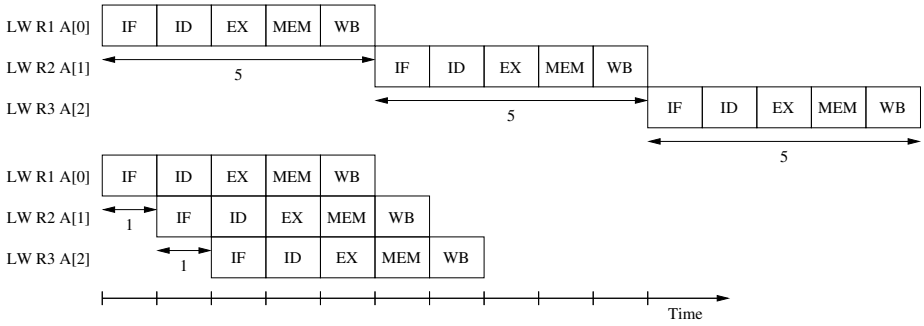


Figure 2.3: Pipelined (bottom) versus non-pipelined (top) approach to load three array elements into three distinct registers using a *load word* (LW) instruction.

2.4.2 Pipelining

First of all *pipelining* was introduced to let multiple instructions execute concurrently, thereby increasing instruction throughput. To this end, execution of a single instruction was split up into a sequence of distinct *stages*, such as: *instruction fetch* (IF), *instruction decode/dispatch* (ID) to interpret the instruction and fetch the operands, instruction *execution* (EX), *memory access* (MEM), and*write back* (WB) of a result to a destination register. The reason that memory access comes separately, after execution, is because often a memory *address* needs to be computed first, which is done in the EX stage.

Each instruction executes one stage per clock cycle, with one instruction executing per stage, so that subsequent instructions can overlap, as illustrated in Figure 2.3. This means the maximum number of instructions that can be executed concurrently, often referred to as a processors *window*, is equal to the number of pipeline stages, or *pipeline length*. By allowing parallel computation of instruction stages, we utilize CPU resources more effectively and allow for higher clock speeds due to shortened and simplified pipeline stages. This can significantly increase instruction throughput (bandwidth) of a CPU, as a faster clock means that more instructions can finish per second.

2.4.3 Super-scalar Execution

The restriction to one instruction per pipeline stage is not strictly necessary. As long as sequential program execution semantics are not violated there is nothing against executing more than one instruction per stage. To handle multiple instructions per stage we need at least two parallel pipelines. The related measure is the *pipeline width*, or *issue width*, which represents the number of such parallel execution paths. An often used term for this kind of architecture is *super-scalar*.

2.4.4 Hazards

Both pipelining and super-scalar execution increase peak instruction throughput. Assuming that the maximum number of concurrently executing instructions per pipeline stage is constant across stages, a processor can theoretically

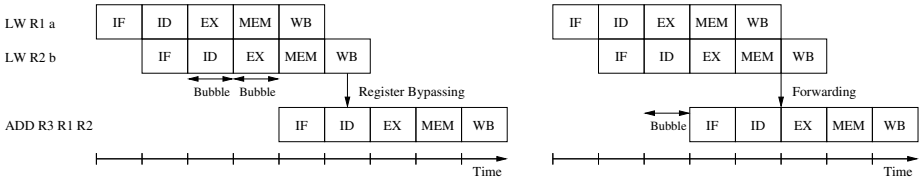


Figure 2.4: Adding two variables, with (right), and without forwarding (left). Note that the left figure assumes the registers can be written and read correctly within the same cycle.

have a window of *pipeline length* $\times$ *issue width* in-flight instructions at an arbitrary moment in time. In practice, that maximum is hardly ever achieved however. This has two main reasons, being the dangers of *data hazards*, and *control hazards*.

Data Hazards

A data hazard, or *data dependency*, occurs if an instruction needs a data item that still has to be produced by some preceding instruction that did not finish execution yet. This can be partially solved by *forwarding* results to earlier pipeline stages as soon as they become available, thereby saving the additional cycles needed to communicate the result through a register. But even using forwarding it can happen that an instruction has to wait for a result to become available. In these cases the pipeline needs to be *stalled* by introducing so called *pipeline bubbles*, instructions that simply do nothing, until the hazard is resolved. To illustrate this, let us compute the sum of two variables, `a` and `b`: Which can be compiled into:

```
LW R1 a       # load variable a from memory into register R1
LW R2 b       # load variable b from memory into register R2
ADD R3 R1 R2  # compute sum into R3
```

The result of executing this code can be found in Figure 2.4, which illustrates the needed pipeline stalls, both with and without forwarding. Even when forwarding the result from the end of the MEM stage to the beginning of the additions EX stage, we need to insert a bubble.

Control Hazards

Control hazards, or *control dependencies*, are even more severe than data hazards. They are introduced by if-then-else branches in a programs execution. If a branch relies on the outcome of some computation, the *branch condition*, to decide which instruction needs to be executed next, and if this outcome is not yet known, it is also unknown which instruction to issue next. Forwarding will not work here, as there is simply no instruction to issue, and thus no instruction to forward to. The address of the branch target instruction is known after EX. But then that instruction still needs to be retrieved from memory during the MEM stage. This introduces three bubbles in our five stage pipeline, as shown in Figure 2.5.

To illustrate the effect of control hazards, let us look at the following `if--then-else` statement, which copies `a` into `b` if `a` is smaller than 255, and sets `b` to 255 otherwise:

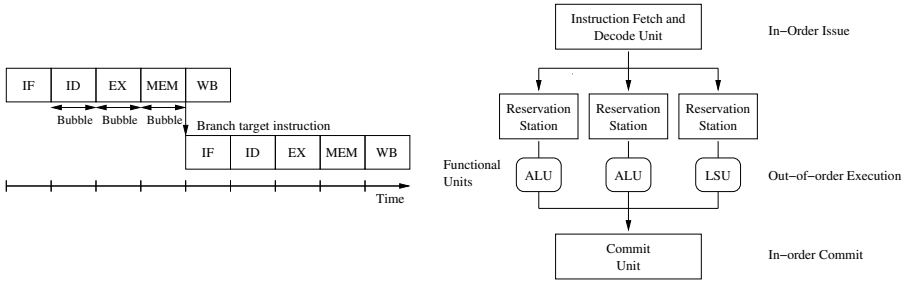


Figure 2.5: Left: three bubbles after a branch instruction. Right: simplified dynamic pipeline with one *load-store unit* (LSU), and two parallel *arithmetic-logic unit*s (ALU).

```
BGEQ R1 256 if-else-label  # branch to else if a >= 256
NOP                        # wait for branch condition
NOP                        # wait for branch condition
NOP                        # wait for instruction to be loaded
ADDI R2 255                # assign immediate value of 255 to b
...
```

Figure 2.6: Stalls due to a control hazard, represented as *no-operation*s (NOP).

```
if (a < 255)
    b = a ;
else
    b = 255 ;
```

Assuming that `a` and `b` are assigned to registers `R1` and `R2` respectively, this would compile into the following MIPS code:

```
BGEQ R1 256 if-else-label      # branch to else if a >= 256
    ADD R2 R1 0                # copy a into b
    B if-end-label             # branch to end of if-then-else construct
if-else-label:                 # symbolic address label (not an instruction)
    ADDI R2 255                # assign immediate value of 255 to b
if-end-label:                  # symbolic address label (not an instruction)
...
```

Furthermore, let us assume that `R1` is already loaded with a value of `a = 512`, so we can ignore any data dependencies. If we try to run this code on our five-stage pipeline, we will need to wait three cycles after executing the first instruction, just to load the instruction at address `if-else-label` as the next instruction. This is illustrated in Figure 2.6.

If `a` would have been smaller than 255, the unconditional branch, `B if--end-label`, would introduce such a delay as well, as it also needs to load the corresponding instruction.

To reduce the impact of control hazards, modern processors try to *predict* the outcome of a branch instruction using so called *branch predictors*, which keep statistics on past branch behavior, and try to predict the outcomes of current branches based on those statistics. The CPU then speculatively executes instructions from the predicted path, just to keep the pipeline filled. If the prediction turns out to be wrong, the speculatively executed instructions need

to be rolled back by *flushing* the pipeline. However, depending on the execution history of a branch, and on the nature of the branch prediction mechanism, in the worst case, we could still have a situation where all branch outcomes get predicted incorrectly. As with pipeline bubbles, this results in wasted cycles and is thus undesired.

Furthermore, *indirect jumps* in program flow are harder to predict. This happens, for example, with dynamic function calls, which are subject to so-called *late binding*. Here, the address of the function code is stored in some variable. This means that, first, the value of the variable needs to be loaded from memory. Next, we can retrieve the instruction at the corresponding address. After this, we are finally ready to start executing the instruction at the target address. This can easily result in lots of unused CPU cycles. Modern CPUs from Intel and AMD are already trying to hide this by speculating on the jump destination (i.e. the pointer value).

2.4.5 IPC and ILP

By now it should be clear that due to hazards it will often be hard to keep the full window of *pipeline length* $\times$ *issue width* instructions entirely filled. To see how well a CPU performs in this respect, an often used metric is *instructions per clock-cycle* (IPC), which represents the average number of instructions that actually commit per clock-cycle, during the execution of some piece of code. The IPC should ideally be as close to a processors issue width as possible.

A related metric is *instruction level parallelism* (ILP), which measures the potential overlap between instructions, i.e. the average number of operations that can be performed in parallel within a given program. Data and control dependencies are a major threat to such ILP. Another threat comes from *structural hazards*, where instructions are competing for a finite amount of CPU resources, like a single load/store unit.

Finally, it should be mentioned that the discussion of hazards was somewhat simplified. In reality pipelines are often longer than five stages, varying somewhere between seven and thirty-one stages. This will often increase the number of bubbles that need to be inserted on stalls due to data and control hazards. Also, memory loads were assumed to be instantaneous, which, as will become clear in Section 2.5, is not the case in reality either.

2.4.6 Out-of-Order Execution

In search for higher IPC scores, *dynamic pipelining*, or *out-of-order execution* (OoO), was introduced. It means that the CPU tries to *hide* pipeline stalls by scheduling instructions that come *after* a stalling instruction. In fact, this way the CPU tries to exploit instruction level parallelism by reordering independent instructions at run-time.

Figure 2.5 shows a schematic of a dynamic pipeline with three major units:

1. An instruction fetch and issue unit responsible for fetching and decoding instructions in program order, and dispatching them to a corresponding functional unit.

2. The functional units execute instructions in parallel and in arbitrary order, assuming their operands are available. The *reservation stations* buffer in-

coming instructions and their operands, and keep track of which operands are available and which not.

3. The commit unit is responsible for ensuring that sequential program execution semantics are maintained, and is able to decide when it is actually save to write a result to a register or to memory. In fact it reorders instructions into sequential program order.

This example pipeline can potentially execute two arithmetic operations and one memory transfer in parallel. Of course, whether this does actually happen depends on whether the instruction fetch and decode unit is able to issue three instructions per cycle, and in what proportion and order the different type of instructions appear. If it can only issue two per clock, this limits the parallelism during the execution stage to a maximum of two instructions, unless we can combine it with a buffered instruction from one of the *reservation stations*. Furthermore, if these two instructions happen to be memory operations, we can only execute one of them, the second one being buffered in a reservation station.

2.4.7 Role of Compilers

Compilers can increase instruction throughput by hiding, and sometimes even entirely removing data and control dependencies. Hiding can be done by reordering instructions, which is known as *static scheduling*, as opposed to the dynamic, run-time scheduling performed by out-of-order processors. By reordering instructions, a compiler can sometimes fill slots that would otherwise contain a pipeline bubble with independent instructions from other locations in the program, for example by moving memory load and branch instructions to an earlier slot.

Loop Unrolling

To reduce the overhead of loops, such as incrementing loop counters and testing the loop condition, a compiler can *unroll* loops. Logically, it will rewrite a loop of the form:

```
for (i = 0; i < n; i++)
    a[i] = a[i] * x + y;
```

into:

```
for (i = 0; i < n - 4; i = i + 4) {
    a[i] = a[i] * x + y;
    a[i+1] = a[i+1] * x + y;
    a[i+2] = a[i+2] * x + y;
    a[i+3] = a[i+3] * x + y;
}
while(i < n)
    a[i] = a[i] * x + y;
```

Loop Pipelining

Another compile-time optimization on loops is *loop pipelining*, which transforms an operation consisting of multiple dependent operations, `f()` and `g()`, on all `n` independent elements of an array `a` from:

```
f(a[0]), g(a[0]), f(a[1]), g(a[1]),..., f(a[n]), g(a[n])
```

into:

```
f(a[0]), f(a[1]), f(a[2]), g(a[0]), g(a[1]), g(a[2]), f(a[3]),...
```

This way, the compiler can make sure that the modified value of `a[0]`, after `f(a[0])`, is available by the time `g(a[0])` needs it, and so on, potentially removing all data hazards entirely. For example, we could further optimize the core loop from our loop unrolling example into:

```
for (i = 0; i < n - 4; i = i + 4) {
    a[i] = a[i] * x;
    a[i+1] = a[i+1] * x;
    a[i+2] = a[i+2] * x;
    a[i+3] = a[i+3] * x;
    a[i] = a[i] + y;
    a[i+1] = a[i+1] + y;
    a[i+2] = a[i+2] + y;
    a[i+3] = a[i+3] + y;
}
```

Thereby hiding a data dependency stall where we have to wait for the result of the multiplication to become available before being able to apply the addition.

2.4.8 Coding Techniques

Loop unrolling and pipelining are examples of static scheduling optimization techniques that can be detected and introduced automatically by a compiler. To explicitly expose more "hidden" forms of instruction- or data-level parallelism, a programmer can employ several coding techniques. By rewriting code into logically equivalent alternatives with higher ILP, a CPU can be pushed towards more aggressive speculative execution. This section describes two such dynamic scheduling techniques, predication and multiple cursors.

Predication

In general, the negative performance impact of control hazards can be considered worse than that of data hazards. The reason for this is that data dependencies can be resolved earlier by means of *register forwarding*, where the result of a given instruction is fed directly into a dependent instruction in some earlier pipeline stage, thereby shortcutting communication of the result through the register file. The result is that data dependency stalls typically consume less cycles than stalls due to branch misprediction, as those need to flush and restart the entire pipeline. Because of this, it can sometimes be beneficial to transform a control hazard into a data hazard by a technique called *predication* [Ros02]. Predication rewrites selective conditions on a collection of values from a naive implementation, with an `if` statement in the loop body, into a more optimal form, which increments a loop counter with a boolean, defined to be either 0 (false) or 1 (true).

For example, to find the positions of all values bigger than 255 in array `a` and store those positions in array `b`, we could write

```
for (i = 0, j = 0; i < n; i++)
    if (a[i] > 255)
        b[j++] = i ;
```

Using predication we can remove the `if` statement by rewriting this code into:

```
for (i = 0, j = 0; i < n; i++) {
    b[j] = i ;           /* always store index i as an exception position */
    j += (a[i] > 255) ;  /* increment exception index j with a boolean */
}
```

Now, the current loop index, `i`, is always written to position `b[j]`, but `j` is incremented by one only if `a[i] > 255`.

Multiple Cursors

A compiler can only extract instruction level parallelism if it exists logically at the source code level. This means that in loops with control or data hazards, where loop unrolling is not effective, it might pay off to extend the loop body with more *independent* code that the compiler or CPU can use to fill up pipeline stalls. A technique to extract this kind of concurrency in case of array iteration, is to split up the array into two or more *distinct* regions, and use a single loop to iterate those regions in parallel by maintaining multiple array indices, or *cursors*.

For example, to hide the remaining data dependency after applying predication in our example from the previous section, we could split up the array in two equally sized halves, and process them independently, as in:

```
int m = n/2;
for (i = 0, j_0 = 0, j_m = 0; i < m; i++) {
    b_0[j_0] = i ;
    b_m[j_m] = i + m ;
    j_0 += (a[i] > 255) ;
    j_m += (a[i+m] > 255) ;
}
```

This type of concurrency is not detectable by the compiler but can often help to fill up all the parallel pipelines in a modern CPU. Theoretically, one can extend this technique to an arbitrary number of cursors, but eventually this will lead to *register spills*, meaning that a lack of CPU registers causes additional cache traffic, due to the inevitable introduction of new variables.

2.4.9 Advanced Architectures

SIMD

In *single instruction multiple data* (SIMD) [Fly72] computing, simple data level parallelism is exploited by means of specialized hardware. The idea is to allow for a single operation, e.g. an addition or division instruction, to be executed on several data items in parallel. To achieve this, SIMD instructions operate on large, special purpose registers that can hold multiple instances of a regular machine addressable data type. For example a 128-bit register that holds four 32-bit floating point numbers. Multiple independent processing units are then able to operate on each data element in parallel.

SIMD only works well on code with little or no branch instructions. It is aimed at processing large streams of consecutive and aligned data of a fixed width type. Typical application areas include scientific computing (matrix multiplication), multimedia and *digital signal processing* (DSP). Many CPU vendors provide specialized SIMD units on their commodity processors. On popular x86

CPUs we have, in chronological order, MMX (64-bit), SSE (128-bit) and AVX (256-bit) from Intel, and 3DNow! from AMD. On PowerPC, SIMD extensions are called AltiVec.

Usage of SIMD still requires considerable programming effort. Although compilers are able to generate SIMD code for simple computational loops, more advanced automatic "vectorization" of high-level code is still an area of ongoing research. To prevent having to write assembly code for utilizing SIMD, developers are typically aided by the availability of *intrinsic functions*, basically macros whose implementation is handled by the compiler.

Looking at an example, the "daxpy" routine from the *basic linear algebra subprograms* (BLAS) library [LHKK79], computes $y = ax + y$. A regular implementation (which an optimizing compiler might still be able to vectorize) could be:

```
for (i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
```

In a SIMD implementation using intrinsics, this would become [1]:

```
for (i = 0; i < num_samples; i += 4) {
    vector_float tmp;                          /* hold intermediate */
    simd_mul_const_vector(tmp, a, x + i);       /* tmp = ax */
    simd_add_vector_vector(y + i, y + i, tmp);  /* y = y + tmp */
}
```

Although the latter looks longer, it does produces faster code, as it translates to a more compact assembly representation, and performs four computations in parallel per iteration.

VLIW

Out-of-order dynamic scheduling as implemented by most modern, high performance CPUs requires a lot of hardware resources to detect parallelism and reorder instructions at run-time. A different approach to achieving such parallelism is taken by *very long instruction word* (VLIW) designs. Just as with super-scalar processors, a VLIW architecture has multiple functional units. However, the hardware is freed entirely from dynamic dependency detection and reordering of instructions. To fill the parallel functional units, a VLIW architecture packages multiple, independent instructions into one very long instruction, often called a *bundle*, of which it dispatches a single instance each and every clock cycle. Once dispatched, a bundle is executed in order. VLIW CPUs belong to the category of *multiple instruction multiple data* (MIMD) processors.

The responsibility of detecting the needed parallelism to create the static instruction bundles is put entirely with the compiler. Because of this dependence on the compiler, it becomes even more important to implement programs in a way that exposes as much instruction and data-level parallelism as possible.

Because the hardware needed for out-of-order execution does not scale well with issue width, VLIW processors start to become especially beneficial when more than four parallel execution pipelines are employed. For example, Intel's Itanium 2 VLIW processor [Int10] uses bundles of six instructions, which is twice the issue width of Intel's out-of-order Pentium 4. One advantage of such an amount of parallelism, is that branch misprediction penalties can often be

[1] For brevity, we assume that `num_samples` is a multiple of four

Processor	Product	Year	Clock rate (MHz)	Latency (clocks)	Latency (nsec)	Transistors (thousands)	Bandwidth (MIPS)	Issue width	Cores
16-bit address, micro-coded	80286	1982	12.5	6	320	134	2	1	1
32-bit address, micro-coded,	80386	1985	16	5	313	275	6	1	1
5-stage pipeline, on-chip I&D caches, FPU	80486	1989	25	5	200	1200	25	1	1
2-way super-scalar, 64-bit bus	Pentium	1993	66	5	76	3100	132	2	1
Out-of-order, 3-way super-scalar	Pentium-Pro	1997	200	10	50	5500	600	3	1
super-pipelined, on-chip L2 cache	Pentium4	2001	1500	22	15	42K	4500	3	1
Dual-core Merom	CoreDuo	2006	2333	14	6	151K	21000	4	2
Quad-core Nehalem	Core-i7 920	2008	2667	14	5	731K	82300	4	4
Hexa-core Westmere	Core-i7 980X	2010	3333	14	4	1170K	147600	4	6
Hexa-core Sandy-Bridge	Core-i7 3960X	2011	3333	14	4	2270K	177730	6	6

Table 2.1: Milestones in CPU technology, looking at Intel product line (extended from [Pat04]).

eliminated entirely by executing both the *if* part and the `else` part in parallel and discarding one of them as soon as the branch condition is known.

2.4.10 Further CPU Trends

We conclude this section with an analysis of CPU trends in practice. Table 2.1 shows the evolution of Intel brand CPUs over the last decades. A few observations can be made: Most notable is the enormous increase in both the number of transistors and the CPU bandwidth (i.e. throughput). CPU latency shows a steady decrease, while clock rates are going up. However, the latency and clock speed trends evolve at a much slower rate than transistor count and CPU throughput. This implies that most of the bandwidth gains do not come from improvements in sequential instruction throughput, but rather from an ever increasing amount of parallelism.

Back in 1965, Gordon Moore already predicted that the number of transistors per CPU would double every 2 years [Moo65]. These added transistors led to more and more complex cores, with all kinds of infrastructure being added to keep pipelines filled (i.e. wide-issue CPUs with reorder buffers and Out-of-Order execution), resulting in higher and higher sequential CPU throughput.

When parallelization techniques for processing pipelines started running dry, CPU vendors shifted focus towards increasing *thread-level parallelism*, by repli-

cating partial or entire CPU cores. The consequence is that, besides exposing instruction level parallelism, application programmers now also become responsible for writing explicitly parallelized programs, to benefit from the added CPU power. In general we can claim that, the more complexity CPU vendors are adding over time, the higher the parallelism becomes. And consequently, the more difficult it becomes for programmers to utilize available CPU resources to their (near) theoretical maximum.

The following subsections provide a brief chronological overview of techniques introduced by CPU vendors to increase potential performance, together with some notes about their impact on application programmers.

Aggressive Pipelining

Table 2.1 shows a clear trend towards deeper (i.e. latency in clock cycles) and wider (i.e. issue width) pipelines. Increases in clock-speed are tightly related to the initial rise in the number of pipeline stages (*"super-pipelining"*). By splitting the processing of instructions into ever smaller stages, CPU vendors were able to keep increasing CPU frequencies. As a marketing strategy, this worked very well, with customers buying into ever increasing clock speeds. A notable peak in this trend was Intel's Pentium4 Prescott architecture (not shown in the table). It had a 31-stage pipeline, accompanied by a maximum frequency of 3.8GHz.

The boosting of CPU speeds through deeper pipelines does not necessarily lead to equal speedups in program execution-times though. This only holds for "CPU-friendly" code. With deeper pipelines, the relative cost of pipelining hazards increases as well, as more and more bubbles have to be inserted. Eventually, CPU vendors realized this, which is marked by a shift back towards shorter pipelines after Intel's Pentium4. These 14-16 stage pipelines still incur a significant penalty for pipeline bubbles though, stressing the importance to write code that is free of hazards.

While Table2.1 shows a stabilizing trend with respect to pipeline length (i.e. number of stages), the issue width, or pipeline width (i.e. number of parallel pipelines), keeps increasing. Intel's forthcoming Haswell architecture even pushes the number to eight parallel pipelines. For programs that exhibit high data- and instruction level parallelism, wider pipelines can contribute to improved instruction throughput (IPC). Furthermore, it enables more aggressive speculative execution. For example, the parallel pipelines could be utilized to execute both the if and else paths of an if-then-else construct, and then committing only that branch which turns out to have its condition satisfied. This way we utilize abundant CPU resources to avoid pipeline bubbles.

Given these trends of deeper and wider pipelines, application programmers and compiler writers are becoming more and more responsible to come up with clever ways to expose sufficient data- and instruction-level parallelism. Several such techniques were discussed earlier in this chapter.

Improved Lithography

Even though CPU vendors have moved back to shorter pipelines, clock speeds keep increasing. Stock Intel i7 CPUs currently ship at speeds up to 3.9GHz [Int]. Recently, chip-vendor AMD even announced breaking the 5GHz barrier with their FX-9590 chip [Adv]. These gains in CPU speed can be attributed mostly

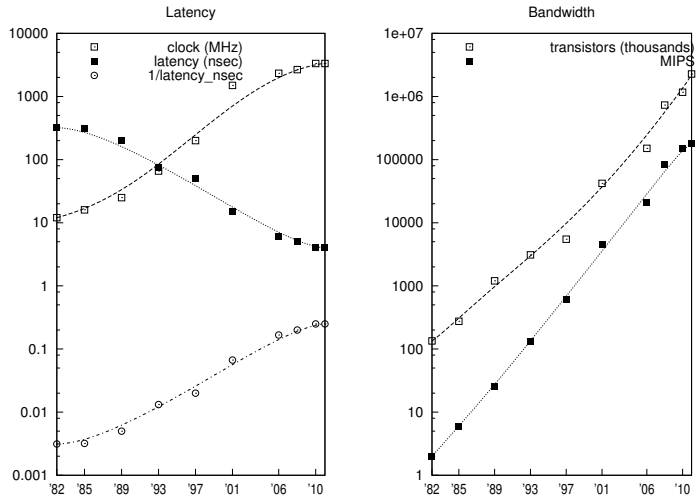


Figure 2.7: Visualization of trends in processor evolution (logarithmic scales).

to improvements in *photo-lithography* (CPU manufacturing technology). Due to increasingly smaller transistor sizes (current Intel i7's having 22nm wide circuit lines [Int]), the physical sizes of CPU cores keep decreasing. Smaller transistors and cores means less heat dissipation and lower power consumption. Also, the smaller a core becomes, the shorter the distances that electrical currents have to travel while flowing through the instruction pipeline. This makes it possible for clock frequencies to be incremented further [DGR+74].

However, as the left graph in Figure 2.7 shows us, the trend of exponential growth in clock frequencies is tapering off. This trend is likely to continue, as it is becoming increasingly difficult (and expensive) to scale manufacturing technologies further down [AHKB00, FDN+01]. Although single-atom transistors have been shown to be feasible [FMM+12], such technologies will be difficult to make cost-effective for mainstream use. A recent estimate based on economic factors expects a minimum lithographic process of around 7nm [Col13].

On the other hand, comparing the CPU latency in the left graph of Figure 2.7 with the bandwidth line on the right (measured in *million instructions per second*, or MIPS), we notice that bandwidth does not show any flattening, increasing at a steady exponential rate. We can also see that bandwidth correlates nicely with the amount of transistors per chip. How CPU vendors utilize these transistors to keep improving CPU throughput is discussed in the following sections.

Simultaneous Multithreading

With the advent of *simultaneous multithreading* (SMT) [TEL95] in later models of Intel's Pentium4, we see the start of a new trend in transistor utilization: rather than adding constructs that try to exploit instruction level parallelism (ILP) within a single thread of execution, entire parts of the CPU, are replicated to improve CPU utilization by allowing multiple threads of execution to run simultaneously on a single CPU, thereby increasing thread level parallelism

(TLP). The most notable hardware changes required for SMT are the ability to fetch instructions from multiple streams within a single cycle, and larger register files to allow each thread to store its own program state.

To benefit from SMT, a software developer can not simply rely on smart compiler and hardware techniques to utilize the added processing power. There is some automatic benefit, in that other processes or the OS itself can now run on the additional "virtual" core, potentially leaving more resources available for the program being optimized. But in general, programmers now have to introduce explicit parallelism into their code, for example by using multiple threads of execution (i.e. *multi-threading*).

Multi-core CPUs

Initial multiprocessing systems were built as *symmetric multiprocessing* (SMP) systems, where two or more identical but discrete CPUs share the same main memory. Reduced transistor sizes, combined with a move towards smaller cores that allow for faster clock rates, has eventually led vendors to spend the ever growing amount of transistors on replication of entire processing cores on a single chip, leading to what we call *multi-core CPUs*, or *chip multi-processors* (CMP). Note that it is perfectly viable to combine SMP and CMP to build a system consisting of multiple multi-core CPUs, with each core often supporting SMT as well. CMP went mainstream in 2006 with Intel's Core Duo, containing two cores, and varies between 2-6 cores in current Intel CPUs. In 2014, Intel is expected to release an 8-core Core i7 CPU based on its latest Haswell architecture. The corresponding server line, named Xeon, is expected to go as far as 15 cores.

For computation intensive tasks, we can even detect a shift towards *many-core* processors, which have an order of magnitude more cores than the typical 2-8 found in regular desktop and server CPUs. These CPUs are aimed at the "*super-computer*" market, where a lot of computational power is needed to solve various simulation, modeling and forecasting style problems. Intel recently released its Xeon Phi processor line, with up to 62 cores, 244 hardware threads and 1.2 teraFLOPS of computing power [Int12b]. An alternative can be found in Tilera's TILE processor line [Til13], where op to 72 identical, general purpose 64-bit cores can be found on a single die.

With CPU latency crawling towards its physical limits, it can be expected that chip vendors will keep focusing on "innovation through increased CPU bandwidth", by putting more and more cores on a single chip. This will put an increasing responsibility on developers to build software in ways that explicitly exposes such thread level parallelism. Such parallel programs are not only harder to write, they are also harder to optimize, as one needs to be able to solve problems like how to distribute work over (a varying number of) cores in such a way that potential contention points like communication buses and caches are utilized most efficiently. This topic is revisited in more detail Section 2.5.4, where this so called *non-uniform memory access* (NUMA), topology is discussed.

Heterogeneous Architectures

Increasing the number of cores by simple copying of existing cores is not the only approach taken by vendors. With hardware becoming more and more complex, common processing tasks are being delegated to specialized hardware.

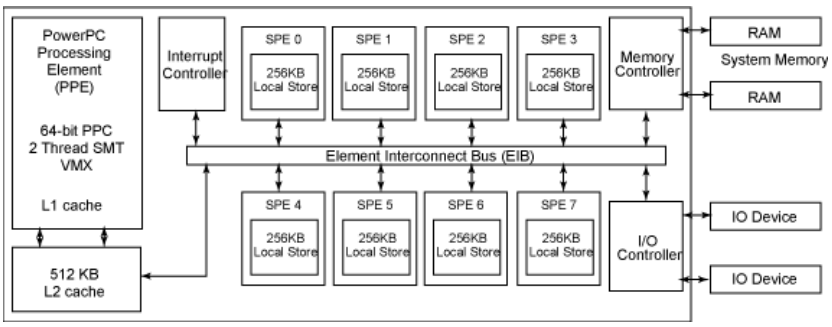


Figure 2.8: Cell Broadband Engine Architecture.

We already saw SIMD in Section 2.4.9 for processing of large data streams as found in scientific, multimedia and DSP applications. But specialized network, cryptographic and even entire *graphics processing unit* (GPU) cores have been added to desktop CPUs as well.

An example of a heterogeneous CPU is the *Cell Broadband Engine* (CBE), or simply "Cell", by IBM, Sony and Toshiba [IBM07]. It tries to bridge the gap between general purpose processors and more specialized stream based computational cores. The CBE consists of a single chip, with one 64-bit general purpose PowerPC CPU core, called *Power Processing Element* (PPE), and eight *synergistic processing elements* (SPE), as illustrated in Figure 2.8. The SPE's are optimized for processing stream data, and are built for a 128-bit SIMD organization, with a preference for single precision (32-bit) floating point arithmetic. They are equipped with 256KB of fast SRAM memory, called *local store*, and cannot access main memory directly. Communication between local and main memory needs to be programmed explicitly by means of *direct memory access* (DMA). PPE, SPE's and RAM are connected by a high speed memory bus, called the *element interconnect bus* (EIB). This architecture makes the Cell well suited for computation intensive tasks, like scientific computing, multimedia and gaming. The Cell CPU can be found in IBM blade servers and in Sony's PlayStation 3 gaming console.

Heterogeneous parallel architectures again put an increasing burden on the programmer, who is now responsible for not only writing parallel code, but to write code for multiple CPU architectures. This often involves programming in low-level SIMD instructions and "micro-managing" data placement over memory. Given the big differences between distinct heterogeneous architectures, writing *generic* code that runs well on all of them is an even bigger challenge.

The OpenCL project [Khr13] aims to provide a standardized programming environment to ease transparent programming of heterogeneous architectures. It allows for the CPU, GPU and various DSP style (co-)processors to be programmed in a unified way, by exposing data- and task-based parallelism through an *application programming interface* (API).

2.5 Hierarchical Memories

The discussion of CPU internals assumed some form of addressable main memory to be available. Technical details and properties like memory size and access

Memory Module	Year	Latency (nsec)	Module width (bits)	Mbits per DRAM chip	Bandwidth MB/sec
DRAM	1980	225	16	0.06	13
Page mode DRAM	1983	170	16	0.25	40
Fast page mode DRAM	1986	125	32	1	160
Fast page mode DRAM	1993	75	64	16	267
Synchronous DRAM	1997	62	64	64	640
Double Data Rate-200 SDRAM	2000	52	64	256	1600
DDR2-533 SDRAM	2004	38	64	1024	4264
DDR3-1066 SDRAM	2006	33	64	8096	8528
DDR4-2133 SDRAM	2013	26	64	16192	17064

Table 2.2: Milestones in DRAM technology (based on [Pat04], with 2003 and 2007 milestones added).

time have been ignored. This section briefly reviews memory technologies and discusses why and how these can be combined to form a hierarchical memory organization.

2.5.1 The "Memory Wall"

Main memory, or *random access memory* (RAM), provides storage of computer data. By "random access" we mean that data is accessible directly, by means of a unique *memory address*. A memory address represents an offset into a sequential address space, typically providing access at the granularity of a byte.

As with CPUs, main memory is built as an integrated circuit on a "memory chip". We can broadly categorize memory chips into two groups: *dynamic* and *static* random access memory, DRAM and SRAM respectively. Both are *volatile*, in the sense that they need electrical current to retain their state.

Main memory is typically built using DRAM, built onto *dual in-line memory module* (DIMM) chips. DRAM stores a memory bit in a single transistor-capacitor pair, which allows for high density and cost-effective mass production. The capacitor holds a high or low voltage, i.e. a 1 or 0 bit. The transistor acts as a switch to allow for reading or changing the capacitor. DRAM needs to be *refreshed* periodically to avoid the capacitors from loosing their charge. It is called "dynamic" because of this refreshing requirement.

Static RAM, on the contrary, does not have such a refreshing requirement. It is built using a fast but relatively complex state storing circuit called a *flip-flop*, which needs four to six transistors. Therefore, it is less dense and more expensive than DRAM, and is not used for high-capacity, low-cost main memory in commodity desktop and server systems. The register file on a CPU die is typically implemented using static RAM.

Table 2.2 summarizes the technical evolution of DRAM technology used for commodity main memory chips. The table refers to SDRAM, or *synchronous dynamic random access memory*, which has nothing to do with SRAM, and is just a synchronized, or "clocked" DRAM variant. When looking at memory latency, which represent the time after which the first bit of a given requested memory address becomes available for reading, we can clearly see that improvements are relatively slow. Once the requested data has become available for reading, it can be transferred at exponentially improving data transfer rates (i.e. bandwidth) though. This means that, with time, larger sequential data accesses are becoming more and more favorable, as these help amortizing access latency. A trend that is supported by a correlated growth in memory storage capacities.

One technological advance that Table 2.2 does not show, is that of *multi-channel memory*. It effectively multiplies the bandwidth of a memory bus, by allowing multiple memory modules to be attached to that bus, each with their own 64 data lines (but shared address and control lines). This allows a single memory controller to access multiple memory modules in parallel, thereby effectively doubling, tripling, or even quadrupling the theoretical maximum bandwidth. With a trend towards memory controllers on the CPU, multi-channel memory has become the de facto standard, with dual-channel being used in desktop systems, and triple- or quadruple-channel being available in memory controllers of high-end desktop and server processors only.

With DDR4, which will be supported by Intel's 2014 high-end Haswell CPUs, a point-to-point topology will be used, where each memory channel is connected to a single module only, and parallel access is regulated by the memory controller. The goal being to simplify timing of the memory bus by moving parallelism from the memory interface to the controller, thereby allowing for faster bus timings and therefore transfer rates. The disadvantage, however, is that for maximum performance, each memory slot will need to be occupied by a DIMM. Contrast this with, for example, a four slot dual-channel setup, where only two DIMMs need to be inserted to benefit from the maximum (two times) bandwidth increase.

Not only are improvements in memory latency lagging behind those in bandwidth, but an even stronger discrepancy can be found in Figure 2.9, where the left side shows relative improvement (compared to 1980 as a baseline) in both CPU and memory latency (the figure actually shows normalized *inverses* of the latency, to signify an improving trend). We see that (the inverse of) CPU latency, i.e. the number of nanoseconds to execute an instruction, has been improving at much higher rates than memory access latency. This has resulted in a roughly hundred-fold improvement of CPU latency versus a meager six-fold improvement of memory access times, over the same period of time. From the processors perspective, i.e. measured in clock-cycles, a memory access is becoming more and more expensive.

The growing discrepancy between CPU and memory speeds is not only present at the latency level. When looking at relative improvements of both CPU and memory bandwidth, as shown in the right-hand side of Figure 2.9, we see a growing gap as well. The relatively poor memory bandwidth has been termed the *von Neumann bottleneck* [Bac78], and is attributed to the word-sized bus between CPU and memory, which is responsible for transferring all memory reads, both data and instructions, and writes.

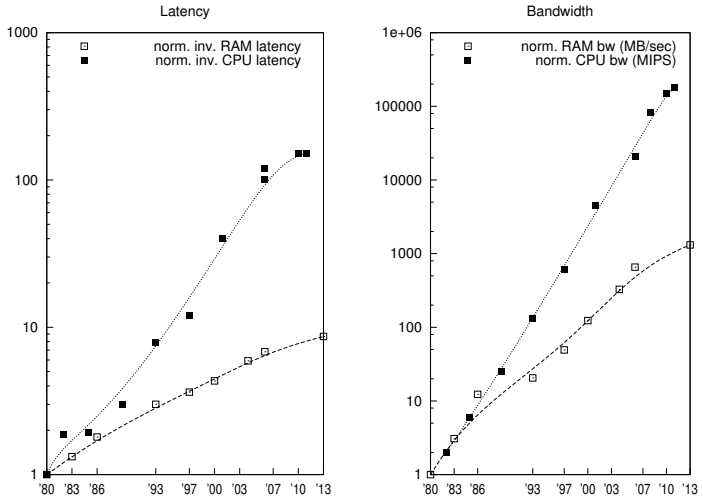


Figure 2.9: Relative improvements in DRAM technology over time (logarithmic scales).

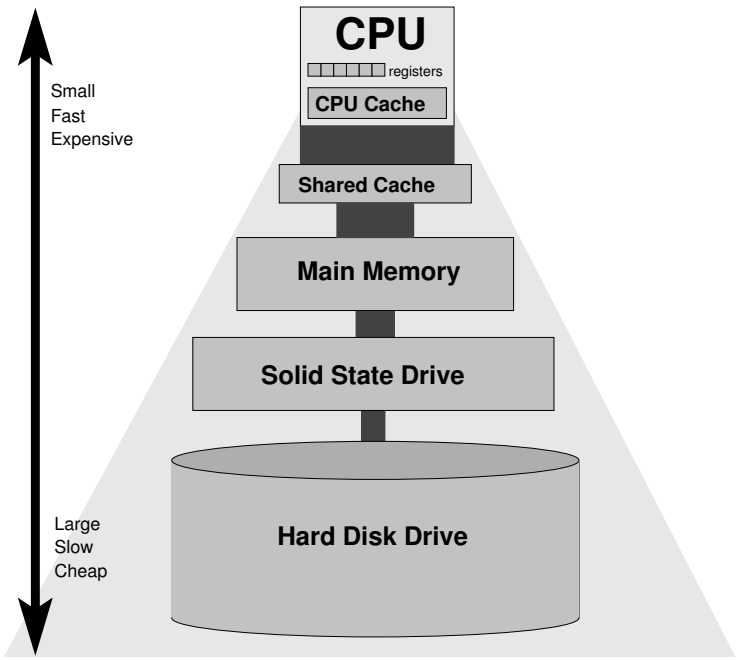


Figure 2.10: Schematic representation of a memory hierarchy.

The disparity between latency and bandwidth within CPUs and memory was already detected and predicted to grow in [WM95], where the term *"memory wall"* was coined. Its existence can largely be attributed to the usage of DRAM memory chips. Not only is DRAM a slower technology than SRAM, but the fact that memory is sold as separate chips, i.e. "modules", implies that it is placed outside the CPU die. Such off-chip communication is both limited in bandwidth, as the CPU and memory modules have to interface with a *memory bus*, and limited in latency, due to the physical distance between CPU and memory that currents have to cover.

2.5.2 Hardware Caches

To hide the negative performance implications of the memory wall, CPU vendors introduced the *hardware cache*, or simply *cache*. Caches are smaller but faster memories that are positioned between the CPU and main memory, typically on the CPU die. Their goal is to provide rapid access to the most frequently accessed memory locations, by maintaining local copies of the data at those addresses. With the memory wall worsening, more caching layers are being added to keep hiding degrading latencies. This is illustrated in Figure 2.10. We can see that for every layer, the closer we get to the CPU, the smaller and faster (but also more expensive) the memory technologies become. Such an organization is called a *memory hierarchy*, and is aimed at making memory look infinitely large, while being as fast and affordable as possible. A summary of some core properties of each layer follows.

register Closest to CPU, and the only memory the CPU can perform computations on. Per core there are typically 4 to 256 registers that hold 32, 64 or even 128-512 bits (for SIMD) each. Registers use SRAM, and have access times of 1-3 CPU cycles.

L1 Cache Fastest (2-10 cycles) on-chip cache, holding 16-64KB of data, often divided into *I-cache* and *D-cache*, for instructions and data respectively.

L2 Cache Larger (128KB-8MB) but slower (10-30 cycles). Usually on-chip, one per core or pair of cores, shared between instructions and data.

L3 Cache A large (1-32MB) on-chip cache that has become common since the advent of multi-core CPUs, where we typically see a single L3 cache that is shared between instructions and data of all cores. Access times are dynamic, depending on sharing state of a cache line, ranging between 40-100 cycles.

main memory Large (1-1000GB) and slow (100-400 cycles) general purpose DRAM storage. Off-chip, so can be configured and replaced independently.

solid-state disk A novel permanent storage technology that can replace or coexist with the slower magnetic disk. It is faster (tens to hundreds of thousands cycles) but also much more expensive and smaller sized.

magnetic disk Large (terabytes) but very slow (millions of cycles) permanent storage.

The rationale behind hierarchical memory organizations is based on the assumptions that a computer system rarely needs to process all the data that is stored on disk at the same time and that only a limited amount of processes is running at a certain moment [Den68]. More specifically, caching relies on the concepts of *spatial-* and *temporal locality*.

Spatial locality : memory addresses near the last referenced one, in terms of physical location, have a higher likelihood of being referenced next.

Temporal locality : a memory address that has recently been referenced has a higher likelihood of being referenced again in the near future than some other arbitrary address.

If we can manage to keep as much of the data that has a high likelihood of being referenced soon as close to the CPU as possible, we have succeeded in hiding much of the performance bottlenecks present in the lower layers of the hierarchy.

CPU caches are fully transparent from a program perspective, as look-ups are handled entirely by hardware. When the processor reads or writes to a memory location, it first checks whether a copy of the data is present in the L1 cache. If it exists, we call this a *cache hit*, and the processor reads from or writes to the cache directly, which is much faster than reading from or writing to main memory. If the given memory address is not cached, we have a *cache miss*, and we need to search the lower layers of the hierarchy until we find the data belonging to given address, so that we can copy it into a cache entry and satisfy the CPU request from there. The fraction of memory accesses that results in a cache hit is called the *hit rate*.

When transferring data to and from main memory, or between cache layers, the minimum transfer unit is a *cache line*, which is typically 32-64 bytes for L1, and 64-128 bytes for L2, but these numbers can vary per architecture. The use of cache lines relates to spatial locality, as for every byte or word that we try to access, we are automatically transferring its surrounding bytes as well.

In order to make room for a new cache entry, the cache may need to *evict* an existing entry. To do this, the *cache replacement policy* applies heuristics that try to predict the entry which is least likely to be accessed again. Often, such heuristics are based on a *least recently used* (LRU) policy, which evicts the cache entry that was accessed least recently. This relates to temporal locality, as the result of LRU replacement is that we keep the most recently accessed cache entries in place.

2.5.3 Virtual Memory

Until now, we treated memory as a large, directly addressable storage space, that can hold all the programs and data that we need. In reality, most of this does not hold. Indeed, memory chips provide a sequential space of uniquely addressable, contiguous storage locations, typically the size of a byte. However, it is undesirable to expose either the finite size of RAM or direct access into it to the program and its writer. Also, providing a more flexible addressing mechanism makes it easier to securely share memory between multiple concurrently running programs.

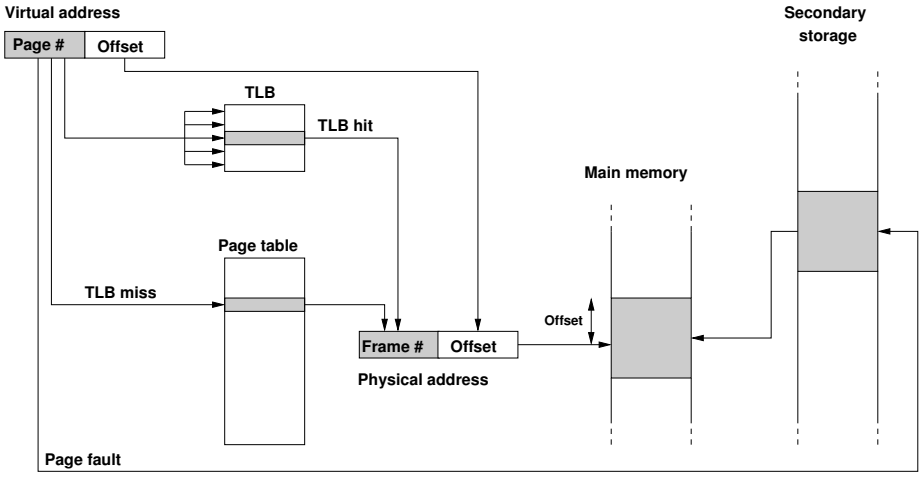


Figure 2.11: Simplified operation of virtual memory.

Such flexible and secure addressing is provided by *virtual memory*. It provides addressing by means of a logical *virtual address*, which is mapped to an actual *physical address* in memory at run time. It allows programs and their data to be located anywhere in main memory, and even for those locations to change over time. Furthermore, program code and data do not even need to appear contiguously in memory, and not all pieces need to be present all the time during execution. The latter implies that programs can have access to a memory space that is larger than the actual physically present RAM.

Virtual memory can be implemented using either *paging* or *segmentation*. Paging partitions the virtual address space as seen by a program into fixed *pages* of consecutive addresses. When a memory piece is allowed to have variable size, we call it a *segment*. A combination of paging and segmentation can also be used.

Virtual memory is a memory management technique that uses a combination of hardware and software. On the hardware side, the CPU provides for dynamic and fast translations of virtual addresses into physical ones, and generates an interrupt in case a referenced page or segment is not in main memory. Software in the operating system is then responsible for processing such an interrupt, usually by bringing in the missing segment or page from disk.

A simplified virtual memory scheme is illustrated in Figure 2.11. When the CPU encounters a request for a certain virtual memory address (VMA), it first checks the *translation lookaside buffer* (TLB) for the presence of this VMA and its corresponding physical memory address (PMA). The TLB is an on-chip cache, which provides fast translation of the most frequently or recently used VMA's into PMA's. In case of a TLB hit, we are done, and the CPU generates the PMA. If the TLB misses, we check the complete *page table*, which resides in memory. The page table either gives us a *frame number*, which maps to the physical location of our page in memory, or it generates a *page fault*, in which case the operating system loads the page from secondary storage. In the latter case, the OS might also have to evict a page from memory in case it's full. As with hardware caches, this is typically done using some form of least recently

used (LRU) policy.

In a typical configuration, the page size is 4KB or 8KB. Most processor architectures and operating systems allow a "large" or "huge" page size to be configured. Such pages can range from 1MB to 1GB on modern x86_64 architectures or even 2GB on Sun Sparc.

Virtual memory can be used to implement *memory mapped files*, where a piece of the memory space is mapped directly, byte-by-byte onto a disk resident file or some other block based I/O device. Rather than having to perform explicit system calls to issue I/O requests through the operating system, memory mapped files allow access to a file as if it were primary storage. The virtual memory manager then automatically brings in pages as they are needed, which tends to be more efficient than I/O through system calls.

For many applications, both program code and the data they modify are small enough to fit at least in main memory, thereby already partly satisfying the locality conditions. For software that needs to analyze or modify large amounts of disk resident data, however, it is important to be aware of the workings and limitations of a hierarchical memory systems. In Section 2.6, we see that disk I/O performance is orders of magnitude worse than memory performance, stressing the need for careful software design.

2.5.4 Memory Access in Multi-processors

To take advantage of the performance potential of today's CPUs, programmers need to write multi-threaded applications that utilize the available CPUs and cores. When multiple CPUs or cores share a single main memory, several complications and architectural properties have to be dealt with.

To ensure program correctness, access to shared data is typically guarded by synchronization mechanisms, which ensure that concurrent accesses get serialized. Such *mutual exclusion* synchronization mechanisms often come with high overheads, and, even worse, have to be overly conservative. Section 2.5.4 introduces *transactional memory*, which has the potential of alleviating this software bottleneck by means of novel hardware mechanisms.

Also at the hardware level, we have to deal with problems like how to scale shared memory accesses, or how to correctly maintain multiple copies of a single piece of data, where each CPU might be modifying it independently in a local memory or cache. These are the topics of Section 2.5.4, where we discuss the prevalent NUMA architecture.

Transactional Memory

Traditional thread synchronization mechanisms have two major disadvantages. First of all, synchronization limits parallelism by serializing access to critical sections in memory. Typically, it holds that the coarser grained the critical sections, the higher the penalty to parallelism. While introduction of a larger number of more fine-grained locks results in additional code complexity, making it harder to write correct programs. The second problem with synchronization is that it has to be applied "statically" (i.e. at development time) and therefore extremely conservatively. Static synchronization is required every time a potential conflict *might* arise, even in case the likelihood of a conflict is low.

Transactional memory, recently introduced by Intel in its Haswell architecture under the name transactional synchronization extensions (TSX) [Int12a], tries to work around these issues by letting programmers mark *transactional sections* in code, and letting the hardware detect "dynamically" (i.e. at run-time) whether threads need to serialize due to a conflict. This lets the processor expose and exploit concurrency in an application that would otherwise be hidden due to overly conservative static synchronization that would turn out to be unnecessary at run-time.

The processor executes each transactional region optimistically, without any synchronization. If a transactional region finishes, or *commits*, successfully, all memory operations performed within that region of code will occur to have appeared *atomically* (i.e. instantaneously) from the perspective of other logical processors. In case the atomic commit mechanism of the hardware detects that a conflict occurred, the optimistic execution fails, and the processor will roll back execution of the transactional region, a process called *abort*. On abort, the CPU discards all updates performed in the transactional region, reverting to a state as if optimistic execution never occurred, and resume execution in a serialized way.

Several types of conflicts may arise during a commit. Most common are conflicting memory accesses between a transactionally executing processor core and another core. Intel's TSX maintains the *read-set* and *write-set* for each transactional region. The read-set is defined by all memory addresses read, while the write-set incorporates all memory addresses written. A conflict then occurs in case some other processor reads a location that is part of the transactions write-set or writes a location that is either in the read- or write-set of the transaction.

The read- and write-set are maintained at the granularity of a cache line, meaning that we can end up with false positives. Also, a transaction may be aborted when reaching certain implementation dependent capacity limits, like the number of accesses in a region. Obviously, aborts lead to wasted CPU cycles and should be kept to a minimum.

NUMA

Traditional symmetric multi-processing (SMP) systems are built using a single main memory, connected to and managed by a single memory controller, which, in turn, is shared by all CPUs. Each CPU is typically equipped with a private cache, which replicates the most recently and frequently used parts of main memory. To tune cache utilization, it is important to try and "bind" software threads to a certain CPU, a concept called *CPU affinity*, support for which is typically provided by the OS.

With multiple copies and versions of data being present at the same time, the need for *cache coherence protocols*, which guarantee correct program semantics under concurrent reads and writes, arises. In early SMP systems, such synchronization had to go through main memory, over the single shared bus, which is already a bottleneck in an environment where multiple CPUs are competing for the scarce memory bandwidth.

To improve scalability, the *non-uniform memory access* NUMA architecture was invented, where the limitation of a single, shared main memory is abandoned, allowing each CPU (*node*), multi-core or not, to have its own local

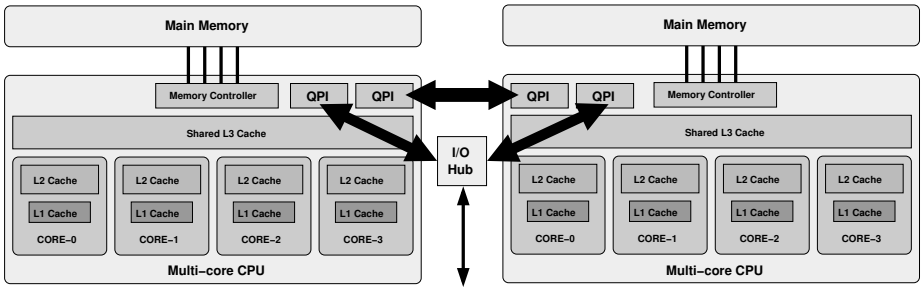


Figure 2.12: NUMA Architecture using Intel Quick Path Interconnect.

Data source	**Latency**
L3 cache hit, line unshared	40 cycles
L3 cache hit, shared line in another core	65 cycles
L3 cache hit, modified in another core	75 cycles
remote L3 cache	100-300 cycles
Local DRAM	60 ns
Remote DRAM	100 ns

Table 2.3: Memory access latency for Intel Core i7 Nehalem [Lev09].

memory controller and attached RAM. Memory of all nodes is aggregated into a logically unified memory, of which the actual access times can vary, depending on the physical location of the data being accessed. To support memory accesses on remote nodes, CPUs need to be connected by means of a high speed communication channel, together with protocols that support transparent remote memory accesses.

An example NUMA system is depicted in Figure 2.12, where we see two multi-core CPUs, each connected to a local main memory through an on-chip memory controller. The CPUs are also connected to each other by means of a point to point link, in this example Intel's QuickPath Interconnect (QPI) [Int09]. These high speed links operate independently from the main memory bus, and are managed by a separate component. Such a setup allows for better scalability than the traditional shared memory systems.

Besides varying memory access latency, a NUMA architecture as depicted in Figure 2.12 suffers from non-uniform cache access (NUCA) as well [Lev09]. The reason for this is that access times to the shared L3 cache depend on the current state of cache-coherence, i.e. whether a cache line being accessed is shared or not, and if so, whether it is modified in the local cache of some other core. Table 2.3 highlights some memory access performance numbers for our two node NUMA example. Note that these are measurements on a single CPU, that depend on things like clock rate and speed of memory used.

As with main memory in SMP systems, the shared *last level cache* (LLC) (i.e. L3 in our example), will become more and more of a bottleneck as the number of cores per chip increases. As the last level cache needs to grow along, even physical wire delays between cores and the large cache area become an issue [KBK02]. Several hardware and hardware aided software techniques are being researched to alleviate problems around shared cache scaling [BS13].

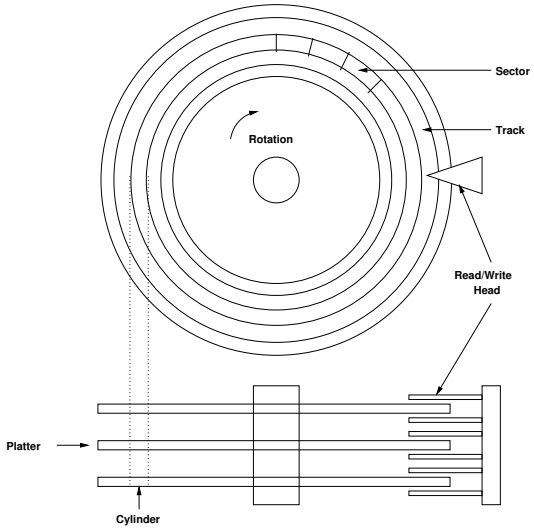


Figure 2.13: Schematic of a hard disk drive.

2.6 Disk technology

A CPU and main memory, which we discussed in the previous sections, are the bare minimum to run a computer program. Typically, users have a need for more permanent, or non-volatile, storage for their data and programs. Furthermore, the data one wishes to manage might exceed the size of main memory. In both scenarios, disk technology is the answer, as it provides permanent and affordable storage for large amounts of data. Traditionally, the magnetic *hard disk drive* (HDD), discussed in Section 2.6.1, used to be the only option. Recently however, the *solid state drive* (SSD), discussed in Section 2.6.2 has become an alternative with distinguishing and attractive properties.

2.6.1 Hard Disk Drives (HDD)

The hard disk drive (HDD) provides for storage and retrieval of digital information, using rotating magnetic disks to store the data. It does this in a random-access manner, meaning that blocks of data can be read or written at arbitrary locations on the drive. A schematic representation of a HDD can be found in Figure 2.13, where we see a *spindle*, responsible for holding and rotating one or more *platters*. The platters contain a magnetic coating, on which data can be encoded using changes in the direction of magnetization. Each platter is organized as a collection of disjoint circular *tracks*, with each track being split into several *sectors*. Matching tracks over multiple platters make up a *cylinder*.

To provide random access to the sectors on disk, the platters are rotated rapidly by the spindle, while a moving arm is used to position the *disk head* over the track containing the sector of interest. The time needed to position the head over a track is called the *seek time*. Once there, on average, the head has to wait for half a rotation of the platters until the given sector moves past the head, something called the *latency* of the drive. This rotational latency depends on the number of rotations per minute (rpm) of a drive. For example, a drive that spins

Product	Year	Capacity (GB)	RPM	Bandwidth (MB/s)	Latency (msec)
CDC Wrenl 94145-36	1983	0.03	3600	0.6	48.3
Seagate ST41600	1990	1.4	5400	4	17.1
Seagate ST15150	1994	4.3	7200	9	12.7
Seagate ST39102	1998	9.1	10000	24	8.8
Seagate ST373453	2003	73.4	15000	86	5.7
Seagate ST3450856	2008	450	15000	166	5.4
Seagate ST3600057	2010	600	15000	204	5.4

Table 2.4: Milestones in hard-drive technology (based on [Pat04], with 2008 and 2010 milestones added).

at 7200rpm, has an average rotational latency of $0.5 \times 1000/(7200/60) = 4.1$ms, and for a high-end drive of 15000rpm we have 2ms.

From a user perspective, what matters is the average *access time*, defined as the sum of average seek time and rotational latency. Table 2.4 lists performance numbers for hard drives over time. Here, the latency column refers to access time, i.e. "real" latency from the end users perspective (but still ignoring data transfer time, which depends on the size of the read and the available disk bandwidth).

Table 2.4 clearly shows a trend where improvements in latency of disk access are stagnating. Only capacity and bandwidth, which both depend on the number of platters and density with which data can be stored on those, keep improving at a steady speed. Still, when we compare the bandwidth numbers with those of main memory in Table 2.2, we easily end up with a factor 50-100 difference.

What is even worse, is that the disk bandwidths from Table 2.4 represent the most optimal scenario: sequential disk access, where no movement of the disk head is involved. When accessing data on disk randomly, bandwidth goes down the drain due to the high latency. For example, reading 4KB pages randomly from disk, the effective bandwidth, assuming a latency of 5ms (i.e. 200 pages per second), is reduced to 800KB per second!

With latencies stagnating, their negative impact on random I/O throughput becomes increasingly worse. One way to alleviate this random I/O bottleneck is to try and amortize the access latency by transferring larger amounts of sequential data, i.e. reading data in large, multi-block sequences For example, at 200MB/s, the transfer times for sequential I/Os of 64KB, 256KB, 1MB and 2MB are 0.312, 1.25, 5 and 10ms respectively, which is still in the order of the 5ms latency. However, the increased transfer granularity would increase effective I/O throughput to 12, 41, 100, 133 MB/s respectively, which is much better than the 0.8MB/s we saw earlier for 4KB blocks.

At the hardware side, several performance improving techniques have been

Product	Year	Capacity	BW (MB/s)		IOPS (x1000)		Latency (ms)	
		(GB)	Read	Write	Read	Write	Read	Write
Intel X25-M 50nm	2008	160	250	70	35	3.3	0.085	0.115
Intel X25-M 34nm	2009	160	250	100	35	8.6	0.065	0.085
Intel SSD 510	2011	250	500	315	20	8	0.065	0.080
Intel SSD 520	2012	480	550	520	50	42	0.080	0.085
Fusion-io ioDrive	2008	320	770	750	140	135	0.026	0.026
Fusion-io ioDrive II	2011	600	1.5K	1.3K	365	800	0.047	0.015

Table 2.5: Milestones in commodity and high-end SSD technology.

introduced as well. We already saw how virtual memory can be used to let main memory act as a cache for the most frequently accessed data on disk. However, most HDDs have hardware caches of 16-64MB themselves, to hide access latency for the most frequently requested pages. Furthermore, a hard disk controller may buffer incoming read and write requests, and process them in an order that results in minimal disk head movement, thereby increasing I/O throughput. A technique known as *request scheduling*.

To increase parallelism, rather than adding multiple disk arms within a single HDD, multiple hard drives can be hooked up into a RAID configuration, as discussed in Section 2.6.3 Also, hard disk drives can be combined with, or even fully replaced by, *solid state drives* (SSD), which do not use any moving parts at all.

2.6.2 Solid State Drives (SSD)

With latency and throughput of the traditional electromechanical hard drive becoming increasingly problematic with respect to the more rapidly improving CPU and memory performance, the need for a better solution arises. Especially with the advent of multi-core CPUs, where each core typically executes an application or thread that has its own I/O needs, the need for solid random-I/O performance arises, as the I/O patterns of each core are likely to differ and therefore interfere. An answer can be found in semiconductor memory that does not lose state when power is turned off: *non-volatile memory* (NVM). Such memory has been around since the 1970s, in the form of EPROM (*erasable programmable read only memory*), and even DRAM chips that were backed up by batteries. However, none of these technologies ever became a reliable and cost-effective alternative to magnetic disk.

Only with the advent of *NAND flash* [Ass95] did the viability of a new mass storage device, capable of replacing the magnetic hard drive, arise. By incorporating NAND flash storage behind a traditional hard disk interface, the *solid state drive* came into existence. Since roughly 2005, when NAND flash process technology surpassed DRAM's 90nm, the technology has become mature and economical enough for mainstream adaptation, delivering what is arguably the biggest boost to commodity hardware performance of the 21st century. Table 2.5 lists some milestones in both commodity and high-end SSDs. A brief overview of SSDs and the underlying NAND flash technology follows. An in-depth treatment of the topic can be found in [MME13].

The most important conclusion from Table 2.5 is that, indeed, flash succeeds

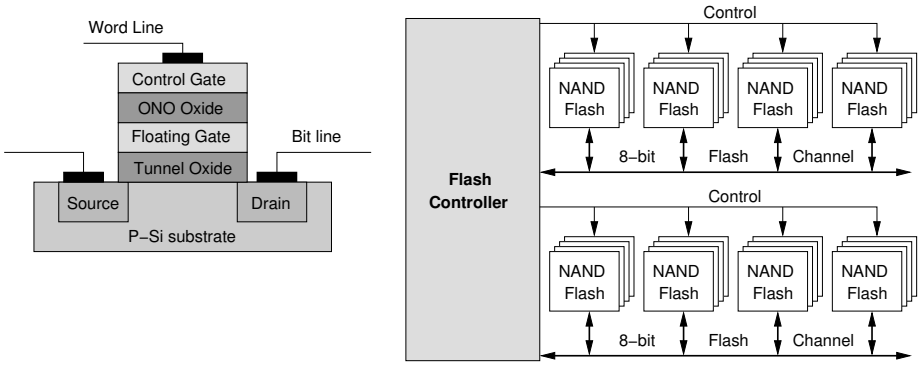


Figure 2.14: NAND Floating gate (left) and NAND Flash SSD (right).

at significantly reducing access latency compared to a traditional HDD (see the earlier Table 2.4). However, we also see that this latency has not been improving. So the novel technology does bring a performance boost, but again it seems hard to improve the latency, while we clearly see improvements in capacity and bandwidth. Besides latency, there is one more advantage of SSDs over HDDs though: SSDs, which use semiconductor technology, are easier to parallelize. The result is that not only has read/write bandwidth been increasing, but also random I/O throughput, i.e. the IOPS column, which represents the number of fixed size I/Os (typically 4-8KB pages) per second. So there is an increasing trend in the number of I/O requests that SSDs can service concurrently, without suffering from random seek latency as seen in HDDs. Random I/O bandwidth is still worse then the optimal (sequential) bandwidth numbers as found in the table, i.e. 50000 4KB IOPS is roughly 195MB/s for the Intel 520, compared to 550MB/s peak sequential bandwidth, but the difference is significantly better than for HDDs.

Looking at other factors than performance, SSDs have two main disadvantages compared to HDDs: their price per GB and their smaller capacity, with SSDs being around 5-10 times more expensive per GB, and HDDs typically having 4-8 times the storage capacity. In many other areas, like *mean time before failure* (MTBF), power consumption, shock resistance, noise, weight and physical size SSDs beat HDDs, positioning them as a solid alternative to HDDs. To maximize the benefits from this new technology, it helps to understand some of its workings and peculiarities, which are discussed in the following sections.

Flash Storage

To retain digital information, NAND flash relies on a *floating gate transistor* (FGMOS) [KS67] with two overlapping gates rather than one, as depicted in Figure 2.14 (left). The floating gate (FG) is entirely surrounded by oxide, thereby isolating it and providing a "trap" where electrons can be stored for years, even when disconnected from power. These trapped electrons influence the conductivity of the tunnel oxide between source and drain, which can then be used to read the state by applying a voltage to the source and sensing either a low or high voltage on the bit line. To program the floating gate, the control gate can be used in conjunction with the source line to either charge (i.e. *program*)

the FG by drawing electrons up from the tunnel oxide, or releasing them to discharge the FG (i.e. *erase*).

We can distinguish two types of NAND flash. In *single level cell* (SLC) NAND each floating gate has only one threshold voltage, thereby being able to represent only one bit of information. In *multi level cell* (MLC) NAND, more than two voltage levels are differentiated, representing more bits of information. For example, four levels to represent two bits of information is very common, but even *triple level cell* (TLC), which differentiates between eight different levels to store three bits per cell, is becoming popular. The advantage of MLC is that it requires the same amount of transistors, while providing several times the storage capacity of SLC, thereby reducing the cost per gigabyte. The biggest disadvantages are that both programming and reading of an MLC takes more time than an SLC. Furthermore, floating gates have the property that they becomes unreliable after a certain number of program/erase cycles, as the insulating material around the gates wears off, reducing its capability to hold a charge. For MLC NAND, this number is much lower (roughly 5000-10000 cycles) than for SLC NAND, which lasts for around 100000 cycles.

Floating gate cells are replicated to build larger memory chips, which can then be used to build an SSD like in Figure 2.14. A bus connects each memory chip to the *controller*, which is responsible for processing and scheduling incoming read and write requests. As with DRAM, each channel uses a double data rate (DDR) interface, and typically multiple channels are employed to boost performance by increasing parallelism.

A flash chip is composed of one or more sub-chips, or dies, each of which might have multiple planes. Each plane is organized as a collection of *blocks* (typically 128KB-512KB), which is the basic unit of operation for the *erase* operation. Each individual block can only endure a limited amount of erase cycles. A block consists of several *pages* (4-8KB), which are the units of operation for read and write operations. Each page consists of a user area, used for data storage, and a spare area, used for status information and *error correction codes* (ECC), which allow for the correction of minor errors during reading. The page size depends on the number of NAND chips on the SSD (i.e. storage capacity) and the number of channels (i.e. parallelism). If these are increased, both page size and throughput increase along. As flash densities increase due to shrinking process technology, we can expect to see a related growth in page sizes. A negative side-effect is that the durability of smaller cells gets worse as well.

Flash Controller

The flash controller is a simple CPU with its own DRAM. It provides the interface to the host for reading and writing data. It is also responsible for interfacing with the NAND storage and performing error correction during reads. One of its most important components is the *flash translation layer* (FTL), which provides a mapping between logical blocks, as seen by the host, and physical blocks, containing the actual data. This is illustrated in Figure 2.15. The FTL has three main responsibilities, each of which is discussed below: *garbage collection*, *bad block management* and *wear leveling*.

Due to the nature of flash chips, bits can not be changed arbitrarily at the individual level. This implies that a page, once written, can not be written to again until it is erased (all bits set to 1, in case of NAND flash). Therefore,

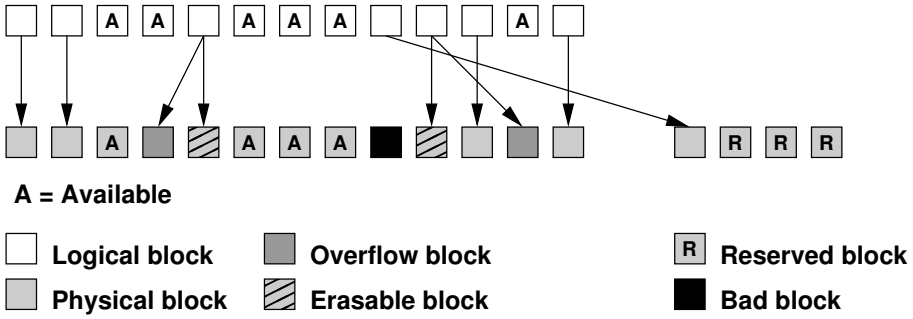


Figure 2.15: Flash Translation Layer (FTL) block mapping.

small writes result in a page to be fully rewritten into a fresh empty page. When this happens, the old page is marked invalid and ready to be garbage collected by the FTL. If no empty page can be found within the original parent block, an overflow block needs to be allocated.

Garbage collection is responsible for reclaiming free space, selecting candidate blocks to be rewritten and erased. To do this, it copies any valid pages in the target block into an already erased block, after which it can erase the target block. This reclaims the space of the non-valid pages in the block being erased. Garbage collection can be performed in the background at times when the drive is idle, or at write time, which is better for write-intense environments where the drive is rarely idle.

A complicating factor for the FTL is the wear of the floating gates in a block with each erase. To improve durability and reliability, the FTL tries to select destination blocks intelligently, aiming to distribute writes evenly among the available blocks, to avoid skew in the wear of blocks. This implies that blocks with a low write count (i.e. those containing read-mostly data) might end up being moved around to allow data that are changed more frequently to be written to those blocks.

Both garbage collection and wear leveling involve moving around data, introducing extra writes that are unrelated to the actual data that the host is trying to write. This phenomenon is called *write amplification*, defined as

$$WriteAmplification = \frac{DataWrittenToFlashMemory}{DataWrittenByHost} \tag{2.1}$$

Write amplification is undesirable, as it consumes extra bandwidth towards the storage layer, reducing effective random write throughput. Sequential writes do not suffer from write amplification.

No matter how smart the wear leveling is, an intrinsic limitation of NAND flash is the presence of *bad blocks*, i.e. blocks containing one or more locations whose reliability can not be guaranteed. These can exist either from factory production errors or due to wear. Bad block management is responsible for identifying these physical blocks and remapping their logical block to a spare physical one. Bad blocks are remapped to reserved spare blocks on the drive, which are made available by *over-provisioning*, i.e. a difference between the physical capacity of a drive and the logical capacity presented to the OS and the user. Over-provisioning is not only used to accommodate for bad blocks,

but also to provide sufficient free space to minimize the negative impact of write amplification during garbage collection and wear leveling.

Above complications put a lot of responsibility and complexity in the FTL, which can become somewhat of a black box to SSD performance, especially if writes are involved. However, in general it is safe to state that writes, especially small ones, should be kept to a minimum whenever possible, with rewrites of large sequential regions presenting the most ideal scenario. Big rewrites also ensure a significant amount of empty space, minimizing negative impact of write amplification. These properties are inherently present in a *log structured file system* [RO92], like Linux' LogFS or the flash specific JFFS. Journaling file systems, like Linux' often used ext3 and ext4 are actually especially ill-suited for flash, as the journaling introduces many small writes.

2.6.3 RAID

To improve capacity, performance and reliability, multiple physical disk drives can be combined into a logical unit called a *redundant array of independent disks* (RAID) (originally *redundant array of inexpensive disks* [PGK88]). In a RAID, data is distributed over multiple physical disks in one of several ways, identified by the *RAID level*, using a combination of *mirroring*, *striping* and *parity* (i.e error correction). RAID's can be built from any disk technology, like HDDs or SSDs, and can be built in hardware or in software at the OS level. Three of the more common performance oriented levels are [CLG+94]:

RAID 0: block level striping without parity or mirroring, stores each block of data on one of the available drives only, i.e. block N goes to drive N modulo the number of drives. This provides improved performance and additional storage but no fault tolerance. Any drive failure destroys the array, and the likelihood of failure increases with more drives in the array. It has the best write performance since it does not need to update redundant information.

RAID 1: mirroring without parity or striping, writes data identically to two drives, producing a "mirrored set". The array continues to operate as long as at least one drive is functioning. It has improved read performance as data can be retrieved from the disk with the shortest access latency at any given moment. Write performance, on the other hand, is governed by the drive with the highest latency at any given time, as redundant copies need to be maintained.

RAID 5: block level striping with distributed parity, stripes blocks over N-1 of N available drives. For each N-1 blocks, a parity block is maintained which allows data to be restored in case of a single disk failure. Typically the parity is simply a XOR of the bits in the N-1 aligned blocks. The parity blocks are distributed evenly over the available drives to avoid the overhead of a single parity drive (i.e. RAID 4). RAID 5 requires at least three drives, and can be used to increase disk I/O bandwidth. If HDDs are used, access latency degrades with an increasing number of drives, unless the angular orientation of the HDDs is kept synchronized.

RAID levels can be nested as well, i.e. in *RAID0+1* a mirror of stripes is built by adding RAID1 mirroring on top of two pairs of RAID0 striped disks. In

RAID1+0, also called *RAID10*, the reverse is being done: two pairs of RAID1 mirrored drives are used to stripe blocks using RAID0.

A consequence of distributing data over disks in blocks (typically 32-64KB), is that, to benefit from the increased bandwidth, all disks need to be employed, meaning that large granularity reads have to be performed that read at least N subsequent blocks, with N being the number of disks. Furthermore, an increase in bandwidth and little or no improvement in latency implies that, to achieve optimal performance, larger sequential chunks have to be read to amortize random access latency. So, in general, RAID also benefits from increasingly large, sequential disk access.

2.7 Conclusion

This chapter provided an overview of recent technological trends in CPU, memory and disk performance. In general, we can claim that in each of them performance is typically improved by adding an ever increasing amount of parallelism. As a consequence, bandwidth improves more rapidly than latency, and the performance gaps between the various layers in the memory hierarchy keep growing, especially in terms of latency. Such latency problems tend to be solved by adding yet another layer to the memory hierarchy, trying to hide as much latency as possible by means of caching.

From a performance engineering perspective, it is important to focus on the general trends, and try to optimize conceptually, with those hardware trends in mind. For example, one should always aim to expose parallelism, both at the single-core and multi-core level, and bear locality of data access and typical caching and synchronization policies in mind.

At the CPU level, developing for performance becomes ever more challenging. Single threaded code should try to expose sufficient data- and instruction-level parallelism to help compilers optimize code and keep modern super-pipelined cores busy. Memory stalls should be minimized by making optimal use of caches on the CPU. The advent of multi-core CPUs with their NUMA style memory access makes this an increasingly challenging. Not to mention the multi-threaded parallelism and synchronization requirements that come with multi-core processing.

At the memory hierarchy level, one should aim to maximize cache utilization in all available layers. Typically, these rely on similar caching strategies, mostly varying in terms of bandwidth and latency, and therefore also in access granularity. In general, random access should be avoided by all means, especially random disk I/O, optimizing for sequential algorithms that amortize latency, transferring only useful data where possible, at appropriate granularity. SSDs are an interesting newcomer in the memory hierarchy, that could act as either a transparent additional layer above magnetic disk, or an explicitly controlled high-speed storage besides the hard disk.

Chapter 3

Relational Database Management Systems

A *database* is a collection of interrelated data. A program used to manage such a database is a *database management system* (DBMS). The main goal of a DBMS is to provide its end-users with a convenient and efficient way to retrieve and manipulate the underlying data. This involves both the definition of structures for the storage of information and the provision of mechanisms that allow the information to be manipulated.

Database systems are used in many application areas. Examples include, banking (customer information, accounts, transactions), airlines (reservations, schedule information), universities (student information, course registrations, grades), telecommunication (call logs, billing), sales (customer, product, purchase information), science (genome, astronomy, pharmaceutical), manufacturing (production, inventories and orders of items in warehouses/stores), and many more.

Nowadays, most mainstream database systems implement the *relational model*, using *relational algebra* as a language to define queries over the data. What follows is an overview of *relational database management systems* (RDBMS), which implement this model. This chapter is intended to introduce or refresh the concepts used in this thesis. It can be safely skipped by someone familiar with database architecture. For a more elaborate overview of the topic the reader is referred to a textbook like [SKS01].

3.1 Relational Model

The *relational model* is due to Edgar F. Codd [Cod70]. Under this model, a database consists of a number of *tables*, built from rows that define a relationship between a set of values. The table itself is often called a *relation*, i.e. a set of relationships, and referred to as r. A table *row* representing a relationship is called a *tuple* or *record*, $t_i = (a_1, a_2, ..., a_n)$. Each distinct value a_i within a tuple is called an *attribute*, i.e. a *column* in a table. The set of permitted attribute values defines its *domain*.

A *relational database* consists of a collection of relations. A *relation schema* $R = (A_1, A_2, ..., A_n)$ defines the layout of each relation r as a list of attributes.

custkey	*name*	*address*	*nationkey*
1	Johnson	Main 31, New York	1
2	Smit	Dam 2, Amsterdam	2
3	Jansen	Plein 23, Den Haag	2

nationkey	*name*	*continent*
1	USA	North America
2	Netherlands	Europe

Figure 3.1: Sample relational database consisting of two tables, `customer` (top) and `nation` (bottom).

A collection of relation schemata defines a *database schema*. A snapshot of its contents at a given point in time is called a *database instance*.

A simple example of a database instance consisting of two tables can be found in Figure 3.1. The topmost table, `customer`, contains three tuples, each representing information about a single customer. The bottom table, `nation`, contains two tuples with country information. Note that this example is simplified for brevity. In reality, one would probably split the relation into more attributes, like `first` and `last` instead of `name`, and split `address` into `street`, `number`, `city`.

Besides regular attributes like `name` and `address`, we also observe two special integer "key" attributes in the `customer` table. The `custkey` attribute defines the *primary key* of the customer relation. A primary key is a minimal subset of attributes that uniquely identifies any tuple of possible relation instances. For simplicity and guaranteed uniqueness, an automatically generated incremental integer ID, like `custkey`, can be used.

The last column in the `customer` table is also a key attribute, but not for the `customer` relation itself. This `nationkey` attribute is a so called *foreign key*, which references a key of some other table. In this example, `customer.nationkey` refers to `nation.nationkey`, which is the primary key of the `nation` table at the bottom. In general, a foreign key of a *referencing* or *child relation*, can only contain values that exist in the related key of the *referenced* or *parent relation*, a property that is called *referential integrity*.

The relational model makes a clear distinction between the logical and physical levels. At the logical level, the model only deals with mathematical sets, where each relation consists of unique attributes and tuples, for which order is irrelevant. At the physical level, however, some form of tuple ordering has to be chosen, either arbitrarily or not. Also, the way in which attributes are stored can vary, and impacts the way we access the database. Physical database design decisions are discussed later in Section 3.3.

3.2 Relational Algebra

3.2.1 Overview

To be able to access, or read, the data in a relational database instance, we need a *query language*. In case of the relational model, this language is called *rela-*

tional algebra, also due to Codd [Cod70]. It defines several *relational operators*, which operate on either one or two input relations and produce a relation as a result. A database *query* then becomes a relational algebra *expression*. Where a relational expression can be either a base relation from the database, a constant expression, or recursively defined by combining one or two subexpressions using the following relational operators.

Select $\sigma_p(r) = \{t | t \in r \wedge p(t)\}$ results in a relation containing all tuples from r that satisfy Boolean predicate p.

Project $\Pi_{A_1,A_2,...,A_m}(r)$ results in a relation containing only the listed subset of the attributes A_i of r. As the result relation is a set as well, potential duplicates are eliminated from the result.

Rename $\rho_x(E)$ Renames the result of expression E to x. $\rho_{x(A_1,A_2,...,A_n)}(E)$ renames the result of expression E to x and the attributes to $A_1, A_2, ..., A_n$.

As relations are sets of tuples, relational algebra furthermore borrows the fundamental set operations, *union*, *difference* and *Cartesian product*. These operate on two relations, r and s, both of which need to share the same schema (i.e. same number of attributes, with compatible types). The result adheres to this schema as well.

Union $r \cup s = \{t | t \in r \vee t \in s\}$ results in a relation containing all tuples from r, s or both.

Difference $r - s = \{t | t \in r \wedge t \notin s\}$ results in a relation containing all tuples from r that do not occur in s.

Cartesian Product $r \times s = \{tu | t \in r \wedge u \in s\}$ results in a relation with a schema which is the concatenation of the schemata R and S. It contains all possible tuples which can be generated by concatenating each tuple from r with every tuple from s. Attributes with the same name have to be renamed using the rename operator.

On top of above fundamental operators, several convenience operators have been added: set intersection ($\cap$), natural join ($\bowtie$), theta join/equijoin ($\bowtie_\theta$), semijoin ($\ltimes$), division ($\div$) and assignment ($\leftarrow$). These do not add additional power to the language. We do, however, discuss two of them that are relevant for the remainder of this thesis: *natural join* and *assignment*. Natural join is one of the most common operations in practice, and assignment is relevant for our discussion of database modifications in Section 3.2.3.

Join $r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1=s.A_1 \wedge r.A_2=s.A_2 \wedge \cdots \wedge r.A_n=s.A_n}(r \times s))$, where $\{A_1, A_2, \cdots, A_n\} = R \cap S$ is the set of shared attributes in the schemata of r and s. The result of a join is equal to the Cartesian product $r \times s$ but only contains those (unique) tuples that satisfy equality on all shared attributes $A_1, A_2, \cdots A_n$. Attribute name collisions can be avoided by prefixing the names with the source relation.

Assignment $variable \leftarrow E$ assigns the result of any relational expression E to a temporary relation variable.

We summarize this section with an example relational algebra query.

$$\Pi_{customer.name}(\sigma_{nation.name=\text{“}Netherlands''}(nation) \bowtie customer) \tag{3.1}$$

Selects the names of all customers that live in the Netherlands from our sample database in Figure 3.1 by first performing the selection on tuples matching "Netherlands" in the `nation` relation, joining the result with customer (on the shared `custkey` attribute), and finally projecting the attribute we are interested in, `customer.name`.

3.2.2 Extended Operations

The base relational algebra operators have been extended with several powerful extra operators [Cod79]. The *generalized projection* allows arithmetic operations in its attribute list. And *outer join* accommodates for missing attribute values, so called *null values*, in either one of the join relations.

A third powerful extension allows for *aggregation operations* [Klu82]. Aggregation takes a collection of attribute values as input and returns a single value, using one of the aggregation functions: *min*, *max*, *sum*, *count* or *avg*. More formally:

Aggregation ${}_{G_1,G_2,\cdots,G_n}\mathcal{G}_{F_1(A_1),F_2(A_2),\cdots,F_m(A_m)}(E)$ where $G_1, G_2, \cdots, G_n$ is a list of attributes to group on, and each F_i is an aggregate function over corresponding attribute A_i. Aggregation partitions the tuples in relation E into groups such that all tuples within a group have equal values for $G_1, G_2, \cdots, G_n$. For each group, it then evaluates function F_i on the multi-set of attribute values A_i within that group.

3.2.3 Database Manipulation

The query language operators we discussed so far can only analyze (i.e. read) data. If one wishes to make changes to the database, a *data manipulation language* (DML) is required. A DML language typically defines three operations: *insert*, *delete* and *update*. With the latter being a convenience operator, which is actually redundant in terms of functionality, as it can be emulated using deletion plus insert. All three are defined using the assignment operator:

Insert $r \leftarrow r \cup E$ where E can either express the result of a query or describe new data to be inserted using a constant expression (e.g. $nation \leftarrow nation \cup \{(4, \text{“}Brown'', \text{“}Market\ 4,\ San\ Francisco'', 1)\}$ to add a new entry to our `customer` relation in Figure 3.1).

Delete $r \leftarrow r - E$ removes all tuples in expression E from base relation r. Note that we can not delete single attributes, only full tuples. To delete the customer named "Smit" one would perform $customer \leftarrow customer - \sigma_{name=\text{“}Smit''}(customer)$.

Update (Modify) $r \leftarrow \Pi_{F_1,F_2,\cdots,F_n}(r)$ updates specific attribute values in relation r, where F_i is either the old value of a tuple in r or a new value if the attribute has to be updated. As an example, $customer \leftarrow \Pi_{custkey,name,\text{“}Shore\ Dr.\ Miami\ 7'',1}(\sigma_{name=\text{“}Smit''}(customer)) \cup (customer - \sigma_{name=\text{“}Smit''}(customer))$ would relocate customer "Smit" to a US address

in Miami. In practice, "database updates" is regularly used to refer to all types of database manipulations, i.e. all of insert, delete and update. Therefore, in this thesis, we use *modify* to refer to tuple alterations at the attribute level.

Some database manipulations are illegal, as typically a database has to adhere to certain *consistency constraints*. For example, a primary key attribute has to stay unique, so an attempt to insert a tuple with duplicate key attributes violates the *uniqueness constraint*. Furthermore, tables are often related by means of a foreign key, as in our example from Figure 3.1. This introduces *referential integrity constraints*, which enforce that we do not insert tuples into a referencing table if they refer to a tuple that does not exist in the referenced table. Similarly, we can not delete tuples from a referenced table if there are still any referencing tuples referring to it.

3.2.4 Structured Query Language (SQL)

Relational algebra lay the foundation for what is nowadays the most widely adopted query language by DBMS vendors, the *structured query language* (SQL). The original language was called SEQUEL [CB74], but the versions that were standardized by both ANSI and ISO refer to it as SQL. There are some differences, but most relational algebra constructs can be found back in queries of the form

```
SELECT expr1, expr2, ... FROM table1, table2, ...
WHERE condition
GROUP BY col1, col2, ...
```

where `SELECT` is (confusingly) relational projection, projecting one or more attribute expressions from one or more input tables. Each relation $table_i$ can be either a base table or a *subquery*, and results in a relational join if more than one input relation is provided. The optional `WHERE` clause implements relational selection, allowing specification of arbitrary Boolean predicates. Finally, the optional `GROUP BY` clause specifies one or more attributes to group on, implementing relational aggregation. For aggregation, the $expr_i$ must be aggregate functions over attributes.

We also have matching data manipulation language (DML) constructs that are self explanatory

```
INSERT INTO table (col1, col2, ...) VALUES (expr1, expr2, ...)
DELETE FROM table WHERE condition
UPDATE TABLE SET col1 = expr1, col2 = expr2, ... WHERE condition
```

Note that compared to their relational algebra counterparts, `DELETE` and `UPDATE` implicitly operate on the table they assign to, and have a selection mechanism built-in. Compared to their relational counterparts, this is more restrictive.

Finally, besides query and data manipulation language constructs, SQL also provides mechanisms to define or change a relational schema (i.e. the base tables) itself. This subset of the language is often being referred to as *data definition language* (DDL). It involves operations like creation and deletion of schema definitions, and their alteration, i.e. renaming them or adding/removing attributes.

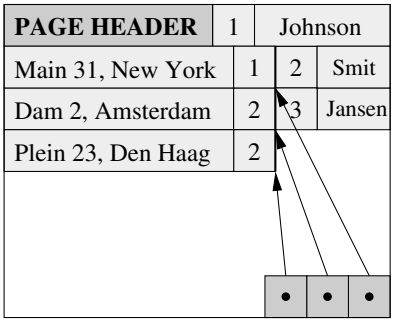


Figure 3.2: NSM storage layout for `customer` relation.

3.3 Storage Management

Relational algebra does not mandate anything about a practical implementation on a computer system. To be able to evaluate queries on a computer, however, one needs to make several design decisions. Two important ones that we discuss briefly are *storage management*, which deals with the actual representation of, and access to, data on disk, and *query processing*, the process of extracting answers to queries from that data. Query processing is introduced briefly in Section 3.4, with more modern techniques being the topic op Chapter 4.

3.3.1 Storage Models

A relational database is typically stored on disk, distributed over several files. A *file* is an abstraction provided by the operating system to identify a list of related *disk blocks* that is used to store objects of arbitrary size. Data is transferred between disk and RAM at the granularity of a fixed size *page*, which represents some constant number of consecutive blocks on disk. For an RDBMS, this means that we need to be able to represent relations in terms of files and pages.

NSM

Traditionally, database vendors have been representing relations on disk using the *n-ary storage model* (NSM, a.k.a. *slotted pages*) [SKS01]. NSM stores n-attribute records contiguously, starting from the beginning of a disk page, and associating an offset with each record in a backward-growing "slot table" at the end of each page. Figure 3.2 illustrates this approach for the `customer` table from our running example relations in Figure 3.1.

NSM performs well when fine-grained access to (almost) entire tuples is desired, as each tuple can be fully retrieved in a single I/O. For this reason, NSM is also a natural choice for update intensive workloads, as insert and delete operate at the tuple granularity. Besides, NSM allows for relatively easy to implement fine-grained locking policies (i.e. page or tuple level), and the slotted page layout can easily leave some unused space in the middle to accommodate for to-be-inserted tuples.

The major disadvantages of NSM are that it wastes both bandwidth and memory space and that it is not particularly CPU friendly. The waste stems from the fact that we transfer and buffer data in RAM at page granularity,

oid	*custkey*
1	1
2	2
3	3

oid	*name*
1	Johnson
2	Smit
3	Jansen

oid	*address*
1	Main 31, New York
2	Dam 2, Amsterdam
3	Plein 23, Den Haag

oid	*nationkey*
1	1
2	2
3	2

Figure 3.3: DSM representation of example `customer` relation.

meaning that we always read and buffer pages containing full attribute data from disk, even though we might only be interested in a subset of those attributes. Furthermore, pages frequently contain unused space in each slotted page, resulting in additional overheads.

Similar inefficiencies can be found higher up in the memory hierarchy, at the level of the CPU and its hardware cache. As small regions of a RAM-resident page (i.e. cache-lines) are loaded into the CPU cache, we again experience potential waste of both bandwidth (from memory) and space (CPU cache capacity). In case we are only interested in partial tuples (i.e. a subset of attributes), spatial locality of the data is compromised, resulting in suboptimal memory bandwidth and additional traffic due to capacity misses in the cache [ADHW99].

DSM

An alternative storage layout is presented by the *decomposed storage model* (DSM) [CK85]. DSM partitions a relation with n attributes vertically into n *binary relations*, as illustrated in Figure 3.3 for the `customer` relation from our example database in Figure 3.1. This results in binary relations consisting of a single attribute column, taken from the original relation, paired with a unique *object-id* (`oid`) that allows us to reconstruct each tuple, thereby maintaining the original relational schema semantics.

Contrary to NSM, each DSM disk page contains only values from a single attribute. Note that, as long as we keep the attribute columns of a table horizontally aligned, there is no need to physically store the object-id. For variable length attributes we still need to store tuple offsets though (or store a var-size attribute in two columns, one containing offsets into the other, containing a sequential data heap). The DSM storage layout of our decomposed `customer` relation can be found in Figure 3.4.

The partitioned nature of DSM allows for fine-grained control over what data is being read from disk and buffered in memory. We can now restrict ourselves to only read pages of those attributes we are really interested in. Similar reasoning can be applied to memory bandwidth and CPU cache utilization, where DSM exposes good spatial locality, which benefits the block-oriented nature of memory transfers and CPU caches.

On the other hand, DSM is not particularly well suited for fine-grained access to (partial) tuples. I.e. in a worst case scenario, where we are accessing all n attributes of a single tuple, DSM requires $O(n)$ page I/Os (compared to 1

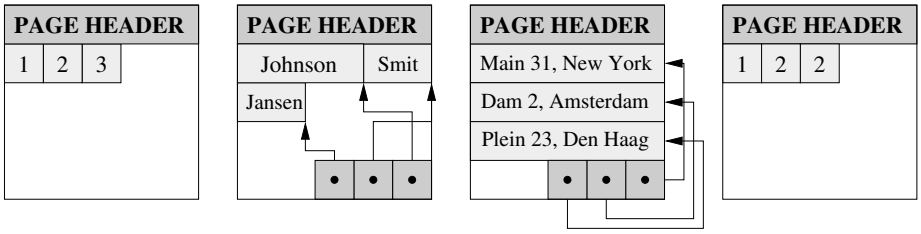


Figure 3.4: DSM storage layout for `customer` relation.

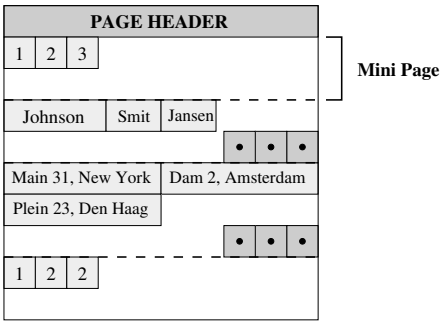


Figure 3.5: PAX storage layout for `customer` relation.

for NSM). As we extend our scan range to include multiple contiguous tuples, eventually the disadvantage compared to NSM will amortize (i.e. after reading the equivalent n pages of NSM data).

The properties of DSM make it particularly well suited for large scans over a subset of attributes columns. However, it is notoriously unfriendly to (naive implementations of) insert and delete operations, as these both require I/O proportional to the number of attributes, i.e. $O(n)$ reads and $O(n)$ writes to insert or delete a single tuple, rather than $O(1)$ in case of NSM.

PAX

The *partition attributes across* (PAX) [ADH02] storage layout can be seen as a hybrid between NSM and DSM. As in NSM, PAX stores the attribute values of a record in the same disk page. However, *within* a page, PAX applies vertical decomposition into so-called *mini pages*, as illustrated in Figure 3.5.

PAX tries to borrow from DSM in terms of memory bandwidth and CPU/CPU cache utilization, exploiting the spatial locality of attribute values within cache lines that make up a mini-page. However, with all attribute data still residing in a single page, PAX does suffer from suboptimal disk bandwidth and memory buffer utilization in scenarios where we are only interested in a subset of attributes. Between RAM and disk, therefore, PAX performs comparably to NSM. The upside of this is that tuple reconstruction cost is equal to that of NSM, i.e. one I/O rather than n I/Os in case of DSM. In terms of CPU efficiency, however, tuple reconstruction using PAX beats NSM [ADH02].

Other, more dynamic, hybrid storage layouts and their trade-offs are analyzed in [ZR03b, ZNB08].

File Organization

The previous sections dealt with ways to *represent* records in a file structure. Another consideration is how to *organize* a set of records (i.e. the tuples in a single relation) in a file. Some prevalent ones relevant for this text are:

Heap organization: a record can be placed anywhere in the file of the table it belongs to, with no particular ordering among records. A simple example is to always append new records at the end of a file. Optionally, one might first try to fill up slots that have been freed by earlier deletions. For DSM this approach becomes more laborious, as each file holds values of a single attribute only, i.e. we need to append or alter in n files, with n the number of attributes.

Sequential organization: records are stored in some sequential order, i.e. the table is kept *sorted*. For this to work, a non-empty subset of table attributes needs to be declared as a *sort key*, or *search key*, according to which the table is kept sorted. Inserts into this file organization need to maintain the invariant that the table is sorted. This means that, in general, we can not simply append at the tail, but need to insert at some fixed position, as governed by the sort key attribute ordering. A major advantage of a sorted layout is that many algorithms that operate on the data can be sped up (i.e. searching, evaluating a range predicate and aggregation).

Clustered organization: this is somewhat of a hybrid between a heap and a sequential organization. As with sorting, a set of *clustering attributes* is chosen. The table is stored in such a way that all tuples with *equal* values on those attributes are stored physically together, i.e. *clustered*, in a subsequence of file pages. Contrary to a sequential file, there is no sequential ordering among distinct clusters though. This organization provides for efficient single-tuple lookups and updates (i.e. *point queries*), but complicates queries involving range predicates, as clustering looses ordering information.

Hash organization: a hash function is computed over a subset of record attributes. The result of this hash function is used to determine in which pages a record is stored. The properties of this organization are comparable to a clustering organization, i.e. good for point-queries and bad for range queries.

Nothing prevents a database system from storing a file multiple times, each according to a different file organization. A technique called *replication*. For example, a table might be stored sorted on two distinct sets of sort key attributes.

3.3.2 Buffer Manager

Since page I/O to and from disk is expensive, database systems try to minimize its occurrence. One way to reduce disk I/Os is to keep as many pages as possible in main memory, trying to maximize the chance that a page we try to access is already available in memory, so that no disk access is required.

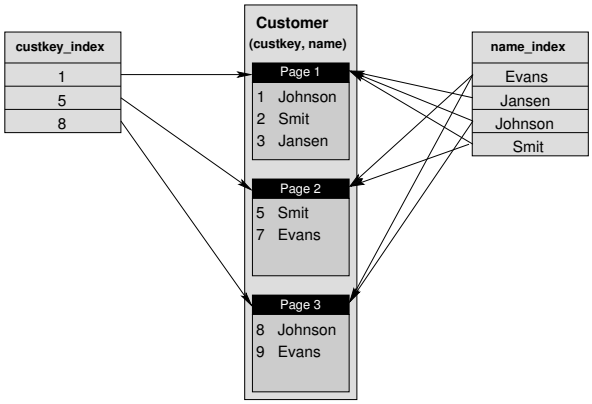


Figure 3.6: A clustered (primary) index on `custkey` and a non-clustered (secondary) index on `name`.

Keeping all database pages in memory is usually not feasible. Therefore, the main memory space for database pages, the *buffer*, needs to be managed intelligently. This is the task of the *buffer manager* (BM), which, in a sense, is similar to that of a virtual memory manager, as discussed in Section 2.5.3.

Besides trying to keep "interesting" pages cached in memory, the buffer manager has to *evict* a certain page in case a page needs to be loaded while the buffer is full. Rather than evicting some random page, a *buffer replacement strategy* tries to do something smarter, like evicting the least recently used (LRU), or most recently used (MRU) page.

Furthermore, the buffer manager should be able to control the writing of pages that have changed (so called *dirty pages*) to disk. The BM can *flush* a page from volatile RAM to disk, making its changes permanent. However, a page that is currently being modified or read from should be *pinned* in memory to avoid it from being evicted.

3.3.3 Indexing

Many queries reference only a small portion of the records in a file. An *index* is an auxiliary data structure that helps retrieving desired records from a table quickly, without examining all records. The primary input of an index is the *search key*, which is a subset of table attributes for which the index speeds up access. The idea is similar to that of an index in the back a textbook, where a list of pointers to relevant pages is printed for each keyword of interest. From a DBMS perspective, the main goal is to minimize the number of I/Os, aiming to only read relevant pages from disk.

Concepts

An *ordered index* maintains *index entries* in search key order, with an index entry being the combination of a specific search key value, together with pointers to each record (or its containing page) with matching search key attributes. A major benefit of ordered indices is that, besides allowing efficient lookup of a single search key (i.e. a *point query*), it also allows for efficient retrieval of

all records with their search key within a certain value range, so called *range queries*.

Orthogonal to the ordering of the index, the table being indexed may also be organized as a sequential file. If the sort attributes of the table match the search key of the index, the index is called a *primary index* (or *clustered index*). If the sort attributes do not match the index key, or the table is simply not stored in a sequential file, the index is a *secondary index* (or *non-clustered index*). Both are illustrated with an example in Figure 3.6, where we see three pages of `customer` data (only showing the indexed attributes). The table is stored in a sequential file organization, ordered on `custkey`. Therefore, the ordered index on the left, `custkey_index` is the primary index. The `name_index` is ordered on `name`, which does not match the sort order of the file. Therefore, this is a secondary index. A table can only have a single primary index, while it can have multiple secondary indices.

A primary index has the advantage that, in general, range queries can be evaluated much more efficiently than for secondary indices. The fact that occurrences of each search key value appear clustered together, and in sequential order, allows each value range to be mapped directly to a set of consecutive pages on disk. Each page in this range (except the two boundary pages) is guaranteed to contain relevant data. On the contrary, for a secondary index, we need to inspect every index entry within our range of interest, and follow its pointers into arbitrary pages, which, due to the incompatible sort orders, may be scattered all over the file. This often results in large amounts of random I/O, into pages that might contain only a single relevant tuple.

Another advantage of a primary index is that we do not need to store an index entry for every possible search key value. This is the case for the `custkey_index` in Figure 3.6, where we only index the minimum `custkey` value within each page. An index with missing entries is called a *sparse index*. Each index entry that does appear in the sparse index should always point to the first appearance of its search key value in the file. For every value not found in the index, we can always infer the two index entries that enclose it, which, given the matching sort order of the indexed file, allows us to pinpoint the relevant pages.

A *dense index* stores an index entry for every search key value. If each index entry can occur multiple times in the indexed file, a list of pointers to each occurrence needs to be maintained, as in our example `name_index`, where names are not unique and may occur in multiple pages. Each index entry therefore may need to store more than one pointer, which is typically implemented by adding another layer of indirection, where each index entry points to a *bucket* of pointers to the actual data.

Note that a secondary index must always be dense, but that the inverse does not hold necessarily, i.e. a dense index may also function as a primary index. In general, a sparse layout fits primary indices, while a dense layout fits secondary indices more naturally. Compared to a dense index, a sparse index is smaller, and also cheaper to maintain under insertion and deletion.

A sequential index organization, as found in our simplified examples in Figure 3.6, might be sufficient in case we are indexing static data. If database manipulations, like insertion and deletion, are involved, the indices of a modified table need to be updated as well. In that case, one can employ specialized data structures that perform better than the linear complexity of maintaining a

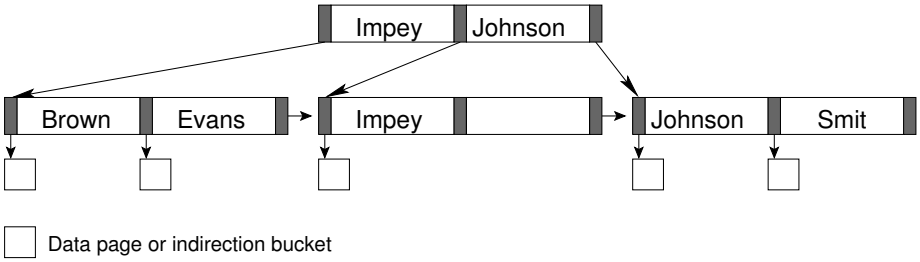


Figure 3.7: Example B$^+$-tree on customer names. This tree has a single root node, with three leaf nodes as children. The middle leaf node is only half full.

flat sequential index. The two most common index structures in RDBMSs are the B$^+$-tree, discussed next, and the *hash index*, discussed after that. We also discuss two more specialized indices, the *join index* and *inverted index*, which are relevant in the context of this thesis.

B$^+$-tree

The B$^+$-tree is the most prevalent ordered index in DBMSs, where it aims to speed up both point and range queries, at the cost of storage space and maintenance overhead under insertion and deletion. The B$^+$-tree is a balanced *n-ary* tree, with a varying, but often quite large number of children per node. It has a *root node*, *internal nodes* and *leaf nodes*. The root may be either a leaf node or an internal node with two or more children. Each node stores $n-1$ search key values and n pointers. The specific value of n represents the *order*, or *branching factor*, of an *n-ary* B$^+$-tree. The n pointers of an internal node point to *child nodes*. For each leaf node, there are $n-1$ pointers that point to the relevant data for $n-1$ search key values. The data pointed to may be either the record or page itself, or a bucket with indirection pointers. The *n-th* pointer in a leaf is used to connect leaf nodes in a linked list, allowing rapid in-order traversal. It is also possible that the leaves do not store data pointers at all, but rather full records themselves. In this case we speak of a B$^+$-tree *file organization*.

An example B$^+$-tree layout is depicted in Figure 3.7. In reality, a B$^+$-tree has a much higher branching factor than the $n = 3$ in our example. The primary aim of the B$^+$-tree is to provide efficient retrieval of data from block-oriented devices, like a disk drive. Therefore, the size of a node is typically a constant multiple of the devices block size. In practice, this means that each B$^+$-tree node can have hundreds or thousands of children, with the actual count of a single node being called its *fan-out*. The fan-out is constrained to stay within $\lceil n/2 \rceil$ and n, except for the root node, which has 2 as the lower bound.

A lookup of value V in a B$^+$-tree is a matter of finding the pointer corresponding to the smallest search key value greater than V, starting at the root node, and following that pointer, repeating the process until a leaf node is found. If the final leaf node has a pointer $K_i = V$, the value was found, with data at matching pointer P_i. This results in a logarithmic upper bound on the lookup complexity of $\lceil \log_{n/2}(K) \rceil$, where K is the number of search keys in the index.

To maintain this logarithmic upper-bound, the B$^+$-tree is kept *balanced*,

meaning that it is automatically reorganized on insertion and deletion to ensure that every possible path from root to leaf has the same length. Whenever an insert would cause a B^+-tree node to exceed n, the node is split into two nodes, potentially recursing up till the root, as long as a parent node overflows due the split in one of its children. Similarly, when a deletion causes a node to drop below $\lceil n/2 \rceil$ entries, a node is coalesced with one of its siblings. Again, such coalescing may have to recurse up till the root. Both operations perform constant work per level of the tree, so that all of insert, delete and lookup end up being $O(\log n(K))$.

Hash Index

An alternative to the sequential file organization can be found in the *hash index*. A hash index maps search key values K to a *bucket*, identified by a number B, using a *hash function*, h, such that $B = h(K)$. Each bucket is a unit of storage, i.e. a page, and is used to store either index entries (search keys with associated pointers) or the actual tuple data. To perform lookup, insertion or deletion on search key K_i, we simply compute $B = h(K_i)$ and either search, insert or delete in bucket B. Note that multiple keys might map to the same bucket, a *collision*, and that searching a bucket in general requires a sequential scan through it.

A proper hash index only keeps index entries in its buckets, with pointers to matching tuples in the indexed table. This non-clustering nature makes it well suited as a secondary index. The term hash index is, however, also often used for a *hashing file organization*, where the buckets hold full tuple data. Because this groups tuples with equal key values together, the hashing file organization is clustered by definition. Although it is clustered, it is not ordered, which makes it ill-suited for range queries. For point queries, inserts and deletions, however, the $O(1)$ performance of hash table operations makes it an attractive indexing structure.

Join Index

A *join index* [Val87] is an auxiliary data structure aimed at speeding up joins between two or more tables. It is motivated by the observation that regular join algorithms were designed and analyzed in isolation, neglecting complexities that arise when multiple joins are combined in a single query. Such multi-way joins are common in, for example, knowledge bases where a DBMS is used to store facts about the world, and an inference engine needs to compute the transitive closure of related facts, distributed over multiple relations. Another example area is that of graph data, where an arc between nodes is represented as the join of two tuples, and long paths need to be analyzed.

A join index is a simple and compact structure that effectively materializes a precomputed join, and can ideally be kept in main memory. It relies on automatically generated integer tuple identifiers, or *surrogates*, to identify and relate tuples. An example is shown in Figure 3.8, where two tables, `CUSTOMER` and `PRODUCT`, are shown, together with two join indices, `JIcsur` and `JIpsur`. The common join attribute is `cname`, and both tables are extended with a surrogate column, `csur` and `psur`, which is unique per relation. Each join index has both a `csur` and a `psur` column. A record in a join index represents a pair of surrogates that link the corresponding `CUSTOMER` tuple with given `csur` and

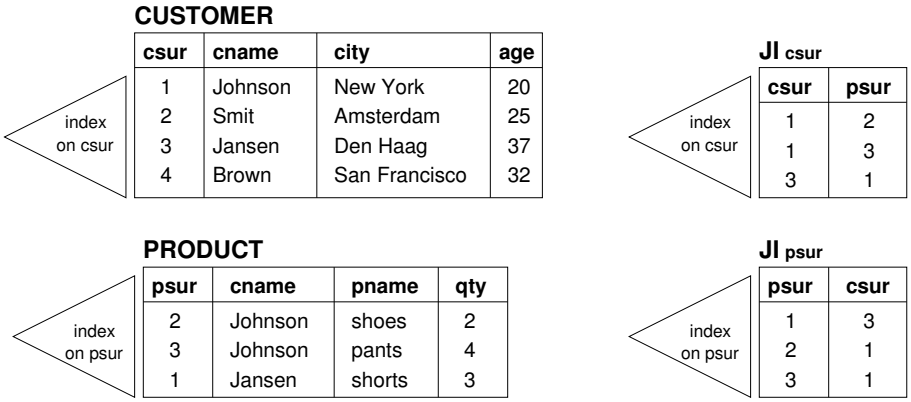


Figure 3.8: Two tables (`CUSTOMER` and `PRODUCT`), connected by two join indices, `JIcsur`, clustered on `csur`, and `JIpsur`, clustered on `psur`.

`PRODUCT` tuple with given `psur`, for every pair of tuples that match on the join attribute `cname`. Both join indices encode exactly the same matching pairs, the difference being that `JIcsur` is clustered on `csur` and `JIpsur` on `psur`, to allow efficient access (through the clustered indices) when navigating in either direction. For example, if we wish to find all the products purchased by "Johnson", having a `csur` of 1, we can quickly find the `psur` values of the products bought using `JIcsur`. If, on the other hand, we wish to find customers that bought shorts, we first find the `psur` of shorts, and then efficiently retrieve `csur` values of the relevant customers from `JIpsur`.

One could argue that the information in a join index could be represented by extending the tables with a "foreign surrogate" column. However, the power of the join index lies in the fact that it is a small and standalone structure, that only needs to store pairs for actual matches. It allows us to efficiently follow join paths over multiple relations, without touching the actual base tables. With the join indices much more likely to fit in RAM than the base relations, this can provide a significant performance gain when evaluating joins.

In Section 4.4.4, we propose a variant of the join index, that does not rely on surrogates, but rather on a static tuple position (offset) for each tuple that does not need to be stored. Furthermore, our alternate representation can be compressed effectively.

Inverted Index

An *inverted index*, also called *inverted file* or *postings file* [ZM06], is an index structure that most closely resembles the index in the back of a book. It stores a mapping from terms (or possibly other forms of content) to their locations in a database file or document collection. It is one of the core indexing structures in most *information retrieval* systems, like search engines and other types of systems that provide keyword search [WMB99, BYRN+99].

An inverted index consists of a *lexicon*, which is a list of all the distinct strings that appear in a collection of documents or database file, typically stemmed and case folded. For every term in the lexicon, the index further maintains a list of pointers, i.e. the "postings list", to locations where the term appears. The

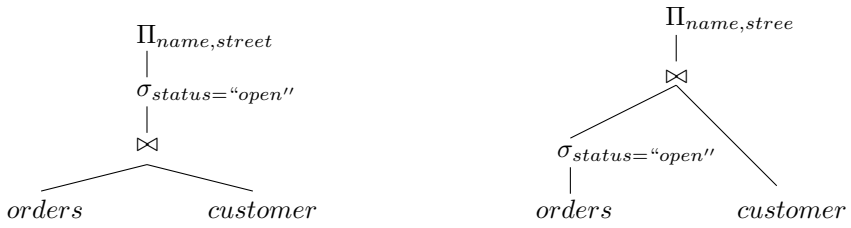


Figure 3.9: Two possible query plans for $\Pi_{name,street}(\sigma_{status=\text{“}open''}(orders \bowtie customer))$.

granularity of each pointer is system dependent. A coarse index might only point to the document containing a term, while a more fine-grained index could point to either line-, word- or byte numbers of each occurrence of that term.

The system can answer a single term query by simply returning the documents that contain that term. Conjunctive Boolean queries of the form *term1* AND *term2* ... AND *termN* can be answered by intersecting the posting lists of all query terms. Similarly, disjunctive (OR) queries can be answered taking the union. Often, the matching documents are *ranked* according to some similarity measure that tries to quantify the relevance of each document to the end user.

Postings lists for each term are often queried and stored sequentially, which makes them good candidates for *compression*, as the differences, or *d-gaps*, between subsequent postings are usually numerically much smaller than the raw postings data. Therefore, d-gaps can typically be stored in less bits than the postings [WMB99, BYRN+99, AM05b, Tro03].

3.4 Query Processing

So far we have introduced the relational algebra query language and some of the mechanisms to store and access relational data on disk. The topic of *query processing* fills in the void between algebra and physical data on disk by describing a generic architecture to support automatic evaluation of arbitrary user queries. Query processing can be subdivided into three core components: a query compiler, a query optimizer and a query execution engine.

3.4.1 Overview

The *query compiler* front-end is responsible for transforming a textual query, as entered by an end user in a language like SQL, into a *query plan*. This involves parsing, syntax checking, and verifying that all relations in the query plan exist and are accessible. A query plan is typically based on an *operator tree* representation of a relational algebra expression, as shown on the left side of Figure 3.9 for SQL query `SELECT name, street FROM orders, customer WHERE orders.custkey = customer.custkey AND status = 'open';`. This tree is a straightforward translation of the corresponding relational algebra expression.

On the right side of Figure 3.9 we see a second query plan, which is logically equivalent to the one on the left, but has the *selection pushed down* below

the join operator. Generating and analyzing alternate query plans is the task of the *query optimizer*, which is supposed to pick the "best" of all possible plans that can be analyzed in reasonable time. This involves searching not only for reorganizations of the operator tree, like selection push-down and join reordering, but also choosing among multiple concrete implementations for each relational operator, such as *hash-join* versus *merge-join* or *table-scan* versus *index-scan* (see below).

The optimizer is supposed to pick the plan that minimizes *query cost*, defined as the estimated time to execute a query. Such cost estimates rely heavily on the algorithmic complexity of each operator and the *cardinality*, or number of tuples, that flows in and out of each. These are then combined with concrete estimates for things like disk access time (i.e. "I/O cost"), average CPU cost per tuple and available memory. The result of the query optimization phase is a *query execution plan*, or *execution tree*, which is an optimized query plan, annotated with the operator implementations picked by the optimizer. Such a query execution plan is ready for execution by the *query execution engine*.

A common technique for evaluating execution trees is *pipelining*. Pipelining evaluates multiple operators simultaneously, feeding the output of one operator, the *producer*, directly as input to its parent operator, the *consumer*, without storing intermediate output relations on disk. In a *demand driven* pipeline, which is the most common, data is being pulled up through the operator tree using a Volcano [Gra94] style *iterator* interface. To implement this interface, each operator needs to provide three calls, *open()*, *next()* and *close()*. Open() and close() are only used for initialization and cleanup respectively. The core functionality is implemented by the next() call, which returns the next output tuple from the operator it is called on. This means that each operator maintains its state between subsequent next() calls. This allows one to pull results "up" through the operator tree in a piecemeal fashion by performing next() calls on the root of an execution tree. Each operator performs recursive next() calls on its children to satisfy its demand for input data, until input is exhausted. Figure 3.10 shows a pipelined execution plan for our earlier optimized query plan in Figure 3.9, but now with concrete operator implementations.

3.4.2 Operator Selection

An extensive overview of query evaluation is out of scope for this thesis. What follows is a walk through several of the choices involved in generation of a final query execution plan, with the query from Figure 3.9 (and its resulting execution plan in Figure 3.10) as a guiding example.

At the lowest level, the leaves of the query plan, we have the relations themselves. This is where raw tuple data is pulled from disk into the operator pipeline. The most basic operator to perform this task is the *table scan*, which outputs all tuples in the table file, in order of appearance. When a selection can be pushed down, it allows us to *filter* the output of a scan to only contain tuples matching the selection predicate. Note that, in general, this does not decrease the number of I/Os performed by a scan, as we still scan all pages. It can, however, significantly decrease the number of tuples being output. The reduction in tuple volume is proportional to the *selectivity* of a selection predicate, with selectivity being the number of tuples that match the predicate (output) divided by the total number of (input) tuples. If a table is sorted (i.e. a sequential file

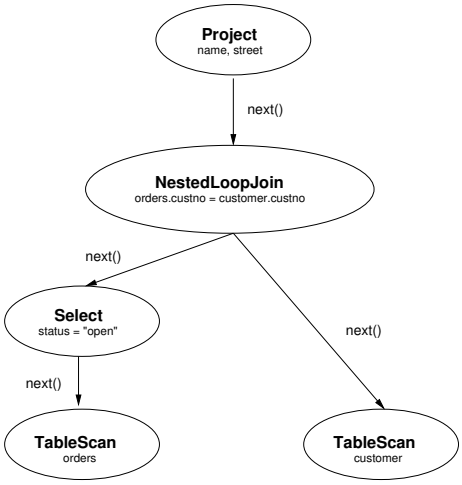


Figure 3.10: Query execution plan for $\Pi_{name,street}(\sigma_{status=\text{“}open\text{''}}(orders \bowtie customer))$.

organization), we might be able to retrieve relevant data using binary search, thereby reducing the search complexity from a linear (in the number of disk I/Os) to logarithmic.

If a table has an index on a selection attribute, it can be used to reduce tuple and/or I/O volumes as well. Note, however, that the presence of a usable index is not guaranteed to deliver a performance improvement, and that index access itself adds some cost as well. In general, point queries benefit most from an index, significantly reducing page I/Os and output tuple count. Range queries benefit from an index as long as it is clustered. In the non-clustered case, a so called *index scan* iterates over index entries in sequential order, but the actual records being pointed to by each index entry are often scattered all over the table, resulting in a random disk access pattern (i.e. “random I/O”). Furthermore, each page might contain only a small number of relevant tuples, with a single tuple per page being the worst case. Therefore, already for small selectivity, a table scan may outperform a non-clustered index scan.

Getting back to our example, we see that there is only a selection on `status = "open"`. Assuming there is no index on the `status` attribute, both `orders` and `customer` can only be accessed using a table scan. The selection is pushed down by the optimizer, resulting in a so-called *scan-select* combination. Decisions to push down are typically made based on the *estimated* selectivity of a selection predicate, which can be calculated with the help of a histogram of attribute values. Assuming that the number of open orders is generally much less than the total number of historical orders, the choice to push down the selection to reduce tuple volume as soon as possible makes sense.

The (filtered) outputs of both scans are then fed into the join operator. Without going into too much detail, the three main approaches to implement a join operator are *nested-loop join*, *hash join* and *merge join*. Nested-loop join basically iterates over tuples in one of the relations, the *outer relation*, and for each tuple iterates over all tuples in the other, *inner relation* to check for a match. In practice *block nested-loop join* and *index nested-loop join* improve

considerably on this inherently slow (quadratic) algorithm. The advantage of nested-loop joins is that they are always available, and have a small memory utilization of two pages at any time.

Hash join is a fast and generic join algorithm that can operate on arbitrary streams of input tuples, as long as the join involves an equality predicate (i.e. no ranges) and a hash function exists on the join attributes. A hash join first builds a hash table over the tuples in one of its input relations, indexed by the join attributes. This relation is typically taken to be the smaller one, the aim being to keep the hash table small, and is called the *build relation*. Hash join then proceeds by iterating over the other relation, the *probe relation*, and for each tuple checks for a match on the join attributes by performing a (constant time) lookup on the hash table. The advantage of hash join is that both build and probe phases are linear in the input size. The main disadvantages are that the hash table can consume a lot of memory, and that its generation is *blocking*, i.e. it blocks the operator pipeline in the sense that hash join can only start generating output in the probe phase, which follows the build phase.

The third kind of join, merge join, combines the linear complexity of hash join with the low-memory pipelined execution of nested loop join. The caveat being that merge join requires both inputs to be sorted. The join operator can than proceed by consuming and joining input tuples using a merge algorithm, where one of the inputs (depending on ascending or descending ordering) is advanced until a match is found. If both inputs can be scanned from disk in sorted fashion, merge join is usually the join operator of choice. In case one or both inputs are not sorted, however, an explicit sort operator would need to be inserted between the merge join operator and any unsorted input. Such an operation, typically *quick sort* in memory or *external merge sort* for large inputs, is blocking and expensive ($O(n \log n)$).

What remains in our example is the project operator on top, which outputs only the desired attributes of each tuple. As with selection, it can be advantageous to push projections down, thereby reducing the size of each tuple, and thereby the volume of data flowing through the operator pipeline. This is especially true for systems that store relations in DSM, so called *column stores*, where we can limit scans to only those columns that are relevant to a given query.

There are many relational operator implementations that we did not discuss. The distinction between ordered and unordered implementations is, in general, a recurring theme, with ordered variants of, for example, aggregation, difference and duplicate elimination allowing for small memory footprints compared to their (often hash based) counterparts. For the interested reader, indexing and access path selection are discussed in [SAC+79], while traditional query optimization is presented in [Cha98]. A thorough survey concerning trade-offs in query evaluation, especially regarding hashing versus sort-merge, can be found in [Gra93].

3.5 Transaction Management

Transaction management is the area that deals with correct and reliable execution of updates to a database. A *transaction* is a sequence of *read* and *write* operations to named data items, that, taken together, represent a single unit

```
read(A)
A := A - 100
write(A)
read(B)
B := B + 100
write(B)
```

Figure 3.11: Example money transfer transaction.

of work. The data items could be of arbitrary granularity, for example a row in a table, a disk page, or even an entire table. A single transaction T_i should transform a consistent database state into another consistent database state. To accomplish this, each transaction is marked by a *start*, and either ends in a *commit* indicating that all its operations completed successfully, or an *abort*, if something went wrong. In case of an abort, all operations performed by the transaction need to be *rolled back*.

An example of a simple transaction would be the transfer of some funds from account A to account B. We assume that account A holds \$600, and account B \$400, so their sum is equal to \$1000. First, the current balance of A needs to be read. Next (if the balance allows it) the transfer amount is subtracted from A, and added to B. As a sequence of read and write operations, a transfer of \$100 dollars would look like the listing in Figure 3.11.

3.5.1 ACID

To maintain a correct database state under concurrent transaction workloads and the danger of hardware failures, a DBMS has to satisfy the four properties known collectively as the *ACID* properties, an acronym for *atomicity*, *consistency*, *isolation* and *durability*.

Atomicity of a transaction ensures that either all operations that make up the transaction are reflected in the database or none at all (i.e. an "all or nothing" commit policy). From a users perspective, a committed transaction appears to be indivisible ("atomic"), and a failed transaction appears to have never happened. If our example transaction would fail before $write(B)$, but after $write(A)$ has completed, the write on A needs to be rolled back, so we end up in the same state as before the transaction started.

Consistency ensures that execution of a single transaction leaves the database in a consistent state, i.e. no invariant is violated before and after execution of the transaction. For example, our earlier funds transfer transaction should not be able to "create" or "leak" money arbitrarily: the sum of accounts A and B should be equal before and after execution of the transaction. Ensuring consistency is considered the responsibility of the application programmer, so we do not discuss it any further here.

Isolation ensures that the *concurrent* execution of several transactions results in a system state that would be obtained if all those transactions would

be executed one after another ("serially"). Concurrent execution of transactions can provide significant performance benefits over serial execution. Therefore, a *concurrency controller* is responsible for ensuring that each transaction is unaware of any concurrently executing transactions. Concurrency control is discussed in more detail in Section 3.5.2.

Durability means that every committed transaction will remain so, no matter what. This means that even after power loss, hardware defects, or system crashes the DBMS is able to return to a consistent state. To defend against power loss, the changes, start, and completion of each transaction have to be written to persistent storage, for example in an append-only file, called the *write-ahead log (WAL)*. Furthermore, it is important to store persistent data with sufficient *redundancy*, for example by using RAID storage or remote replicas of a DBMS. A persistent WAL file should be sufficient to either *redo* any changes that were written to the log but failed to commit to table storage, or *undo* the partial changes of a transaction that failed to commit in its entirety.

3.5.2 Concurrency Control

Conflicts

When two or more transactions are executed concurrently, their individual operations are interleaved. A *history* models this interleaved execution as a linear ordering of reads and writes to data items. Two operations in such a history are said to *conflict* if they involve the same data item but originate from distinct transactions and at least one of the operations is a write. A history is said to be *serializable* if the committed transactions and their effects are equivalent to those of a "serial history", where all the transactions are executed one after another. In general, a history is serializable if we can rewrite it into a serial history by swapping conflict-free operations (i.e concurrent reads or writes to distinct data items).

If we use $r_i(x)$ and $w_i(x)$ to indicate a read and a write of data item x from within transaction i, we can represent the history of our funds transfer transaction from Figure 3.11 running in isolation as $r_1(A), w_1(A), r_1(B), w_1(B)$. Running two such transactions concurrently, many histories are possible, for example $r_1(A), r_2(A), w_1(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$, which has two conflicts. With $w_2(A)$ coming after $w_1(A)$, we have a *write-write conflict*, as the result of $w_1(A)$ is immediately overwritten and thereby lost. In the updates to B we see a *read-write conflict*, or *dirty read*, as $r_2(B)$ reads uncommitted data written by $w_1(B)$.

Isolation Levels

To maintain correct operation of a DBMS under concurrent workloads, a *concurrency controller* is responsible for detecting and dealing with conflicts. This is typically done by guarding the access to a data item with a *lock*, and enforcing a certain execution order that guarantees (partial) serializability. A transaction can request either a *shared lock*, used for read access, or an *exclusive lock*, which can be used for either read or write access. If a data item is already locked by

some other transaction, the requesting transaction has to *wait* until the lock is released, unless both require shared access.

Constraining the execution of database access operations typically means reduced performance, as concurrency is hampered. Therefore, most vendors allow the isolation requirements of their DBMS to be relaxed, compromising serializability for better performance. This has resulted in the standardization of the four ANSI/ISO *isolation levels*, allowing for increasing levels of "read anomalies": *serializable*, *repeatable reads*, *read committed*, and *read uncommitted*.

Serializable This is the highest isolation level, were all reads and writes are locked until the end of a transaction.

Repeatable reads is similar to full serializability, except it does not manage range-locks, which allows so-called *phantom reads* to occur. This means that a single transaction, which selects a range of tuples matching a certain predicate, may see different results in the event that a second transaction inserts new rows satisfying that predicate between the first and second evaluation of the selection.

Read committed keeps write locks till transaction end, but lets go of read locks as soon as a read has completed. This provides the guarantee that any data read is at least committed. However, if a transaction reads a single data item multiple times, no promise is made that it will find the same data.

Read uncommitted is the lowest isolation level, which allows for *dirty reads*, i.e. reads of data items that are not yet committed.

Multiversion Concurrency Control

An alternative to lock-based serialization protocols can be found in *multiversion concurrency control (MVCC)*. MVCC tries to avoid a scenario where readers of a data item have to wait for writers of that data item to release a lock. This is achieved by allowing multiple versions of a data item to coexist at the same time, so that concurrency and performance can be increased by simply generating a new version of a data item each time it is written. Reads are then allowed access to any of the last relevant versions without the need to lock. Such a strategy can significantly boost the performance of read-heavy workloads.

A particularly popular implementation of MVCC can be found in *snapshot isolation (SI)* [BBG+95], or "optimistic concurrency control". Under snapshot isolation, each transaction appears to operate on a private *snapshot* of the database, taken at the start of the transaction. It is allowed to freely change this snapshot under the "optimistic" assumption that it operates in full isolation. The transaction is only allowed to commit if none of the data items it wrote, i.e. its *write-set*, has been changed externally since the transactions snapshot was taken. This policy is called *first-committer-wins*, as transactions trying to commit a conflicting write have to be aborted.

Several vendors have migrated away from lock-based protocols in favor of snapshot isolation. Examples including Oracle, PostgreSQL and Microsoft SQL Server (version 2005 and later). Some of them even refer to snapshot isolation as "serializable" mode, which is not entirely correct. Although snapshot isolation

does not suffer from any of the read anomalies discussed in Section 3.5.2, most implementations do suffer from an anomaly called *write skew*. Recent research has shown, however, that workarounds exist that make snapshot isolation fully serializable [CRF08].

Write skew can only occur in case two data items, A and B, are consistent with a constraint $C()$. Suppose T_1 reads both A and B, then T_2 reads both A and B, writes A, and then commits. Next T_1 writes B and commits. We now have a danger of violating constraint $C()$. This is most easily illustrated with an example. Assume that we are again dealing with two accounts, A and B, this time satisfying the constraint that one of them is allowed to have a negative balance, but the sum of their balances needs to be non-negative, i.e. $A + B \geq 0$ If both accounts start out with \$100 dollar, and T_1 deducts \$200 from A while T_2 deducts \$200 from B, both transactions see a consistent state and will not fail to commit due to a write-write conflict. However, after both transactions have committed, the database is left in an inconsistent state.

3.6 Application Domains

Broadly speaking, database systems are used by two classes of applications, ones that are transaction-heavy, i.e. *online transaction processing* (OLTP) and those that are more analytic and data-intensive in nature, i.e. *online analytic processing* (OLAP) systems. OLTP systems focus on achieving good throughput on highly concurrent workloads involving many database updates. Typical example areas include banks executing financial transactions and web shops processing customer orders. Analytic systems, on the other hand, analyze large quantities of data ("data warehouse"), with the intent of discovering new insights from that data. Application areas include *business intelligence*, *decision support*, and *knowledge discovery*. These areas call for efficient support of complex, ad-hoc queries that are often long-running due to the amount of data analyzed. Data is usually not updated in real time, but "refreshed" in bulk at routine intervals.

In the past, both OLTP and analytic workloads were conducted on traditional business oriented database systems, designed under the "one size fits all" philosophy. These systems provide row-based storage (NSM), B^+-tree indexing and ACID support for transactions. Given that, on one hand, data volumes are ever increasing, while, on the other hand, performance gaps in hierarchical memory systems keep worsening (see Chapter 2), support for a one-size-fits-all paradigm starts to fall apart [SC05]. In Chapter 4 we therefore introduce Vectorwise, a column-oriented DBMS that aims to maximize performance on data-intensive workloads. In Chapter 6 we then show how to add transactional update support to such a read-optimized system.

Chapter 4

Vectorwise: a DBMS on Modern Hardware

4.1 Introduction

Vectorwise started out as an experimental database engine, under the name MonetDB/X100 [BZN05, ZBNH05], within the Dutch scientific institute CWI (Centrum voor Wiskunde and Informatica). This research project was motivated by an observation that traditional database management systems (DBMS) perform poorly in terms of CPU utilization [ADHW99, PMAJ01, ADHS01]. The aim of MonetDB/X100 was therefore to build a novel database engine, from scratch, capable of achieving excellent performance on data-intensive workloads by taking advantage of modern hardware. Building on the lessons learned from its predecessor, the columnar main memory DBMS MonetDB/MIL [Bon02, Man02, Mon], X100 aims not only to improve in terms of CPU and cache utilization, by means of a **vectorized execution model**, but also to scale out of memory by employing efficient **compressed column storage**. Furthermore, to support efficient updates and transactions against compressed columnar data, X100 makes use of **positional differential update** techniques that aim to provide transactional update support with minimal negative impact on the performance of read-only analytic queries.

Since its introduction in 2005, research around MonetDB/X100 has not only resulted in a multitude of publications and the Ph.D. dissertation of Marcin Żukowski [Żuk09], but also in commercial successes through a CWI spin-off called *Vectorwise*, which was acquired by Ingres Corp. in 2011. Shortly after this acquisition, Ingres was rebranded to Actian Corp, which initially sold the descendant of MonetDB/X100 under the name Vectorwise, but changed the naming to Actian Vector recently. In the remainder of this thesis, we stick with the name Vectorwise, even when referring to the pre-acquisition MonetDB/X100 research prototype. Under the Actian umbrella, Vectorwise has consistently been able to deliver top scores on the industry standard TPC-H benchmark [Tra02].

This chapter is intended as an overview of the Vectorwise architecture and the research around it, some of which is presented in more detail in following chapters. We start with a motivation and introduce the vectorized execution

DBMS "X"	MySQL 4.1	MonetDB/MIL	Vectorwise	hand-coded
28.1s	26.6s	3.7s	0.60s	0.22s

Table 4.1: TPC-H Query 1 performance on several systems (scale-factor 1).

model in Sections 4.2 and 4.3 respectively. In Section 4.4 we then introduce ColumnBM, Vectorwise's columnar buffer manager, together with storage and indexing techniques. Section 4.6 briefly touches upon the integration of MonetDB/X100 into the Ingres SQL front-end and summarizes further research conducted in the context of Vectorwise. In general, Vectorwise focuses on improving performance of scan-heavy queries. In Section 4.5, however, we give a short introduction to one of the core contributions of this thesis: **positional update handling** in Vectorwise. We end this chapter with related work, from both industry and research, in Section 4.7.

4.2 Motivation

Research has shown that the computational power of traditional database systems on data-intensive workloads is relatively poor [ADHW99, Ros02]. We illustrate this observation with experimental results from earlier work around Vectorwise [BZN05, ZBNH05]. Table 4.1 shows execution times of TPC-H query 1 on two traditional database systems, "X" and MySQL 4.1, compared against MonetDB/MIL, Vectorwise and a hand-coded C implementation as a baseline. TPC-H is a standardized benchmark for analytic database systems, which, due to its artificial nature can be scaled to arbitrary sizes. The results from Table 4.1 are for scale-factor (SF) 1, which boils down to roughly 1GB of raw data. The query itself can be found in Figure 4.1, where we see a simple scan-select style plan with an aggregation on top. The plan contains no joins, is not hard to optimize and is CPU-bound on all systems, so that it measures raw expression evaluation performance. The selection selects almost all tuples, 5.9M out of 6M, and the final grouping for the eight aggregations results in only four distinct groups, which can be managed in a cache-resident hash table that does not introduce any disk-spilling I/O.

The execution times in Table 4.1 clearly confirm that traditional DBMSs perform poorly in terms of computational power. MonetDB/MIL does a much better job, but is still roughly 6 times slower than Vectorwise, which gets close to the ideal baseline of a hand-coded plan.

A more in-depth analysis of the traditional DBMSs shows that only about 10% of the execution time is spent on "useful" work, i.e. computations related to evaluating the expressions present in the query plan. Most of the remaining time can be attributed to overheads introduced by a *tuple-at-a-time* evaluation of the Volcano [Gra94] iterator pipeline [BZN05]. First of all, much time is spent in function calls that navigate a single NSM record to copy out relevant attributes. Similarly, per-tuple hash table insertion adds another significant function call overhead component. And finally, it is not only the function call overheads themselves, but the fact that per-tuple calls hide data parallelism from the compiler, which results in inefficient sequential code that can not be loop-pipelined.

```
SELECT
   l_returnflag,
   l_linestatus,
   sum(l_quantity) as sum_qty,
   sum(l_extendedprice) as sum_base_price,
   sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
   sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
   avg(l_quantity) as avg_qty,
   avg(l_extendedprice) as avg_price,
   avg(l_discount) as avg_disc,
   count(*) as count_order
FROM lineitem
WHERE l_shipdate <= date '1998-09-02'
GROUP BY l_returnflag, l_linestatus;
```

Figure 4.1: TPC-H Query 1.

All in all, these design decisions, often made long ago (i.e. 70s-80s) and based on outdated assumptions about computer hardware, are causing traditional DBMS systems to perform poorly in utilizing the computational power of modern CPUs. For example, it has been shown that most of these systems score badly in terms of completed CPU instructions per clock (IPC) [ADHW99, PMAJ01], suffering heavily from data- and control-dependencies [Ros02]. The negative effects of such dependencies are further inflated by poor cache utilization, often failing to exploit locality in both instruction and data access-patterns [ADHS01].

MonetDB/MIL, or simply MonetDB, which is still being developed at CWI as open-source DBMS [Mon], uses a different approach to query execution. Its MIL algebra [BK99] minimizes interpretation overhead by providing execution primitives that process data *column-at-a-time* rather than tuple-at-a-time. MonetDB stores relations in a vertically decomposed way (DSM), where columns of attribute values end up as linear arrays in memory. Algebra operators are then implemented by *primitives* that iterate over such an array, applying a fixed operation to it. Such loops expose good instruction code locality, and allow modern compilers to make use of SIMD instructions and loop-pipelining, allowing high IPC scores to be achieved. However, full materialization of intermediate results from each primitive effectively constrains this column-at-a-time execution model to main memory scenarios, limiting its scalability. Even materialization into memory introduces a significant performance bottleneck, which is illustrated by the factor six performance difference in Table 4.1 between MonetDB/MIL and Vectorwise, which, as we will see in the following section, does not suffer from a full materialization bottleneck.

4.3 Vectorized Execution

The main goal of Vectorwise is to obtain the low overhead, column-wise query execution of MonetDB, but without the overheads associated with full materialization. To achieve this, it employs a *vectorized execution engine*, which modifies the traditional Volcano iterator pipeline to pass around *vectors* of at-

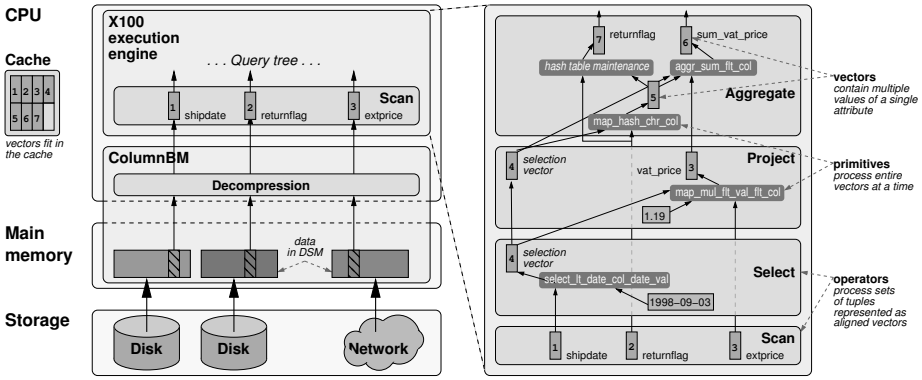


Figure 4.2: Vectorwise architecture overview and execution plan example.

tribute values rather than single tuples. These vectors should be large enough to effectively amortize function call overheads, but small enough to be able to keep intermediate results *in-cache*, so that only cache resident data is passed between operators, effectively avoiding costs of materializing into memory. Besides, Vectorwise aims to design operator algorithms so that they restrict random memory access patterns to regions that fit the CPU cache as much as possible. For example, in the context of hashing, an important operation in operators like join and aggregation, vectorized and cache-friendly techniques are investigated in [ZHB06].

The vectorized architecture is illustrated in Figure 4.2, where on the left we see vectors of DSM data being fed from (compressed) buffer manager pages into the execution engine, which interprets a set of horizontally aligned column vectors to represent a collection of tuples. On the right we see an example vectorized execution plan for our TPC-H Query 1 from Figure 4.1. The scan, select, project and aggregation are organized in a pipeline where each operator consumes and produces column vectors. High level operator logic is shared for all data types, predicates and arithmetic functions. The specifics are handled by *execution primitives*, simple iterative computational kernels built from CPU efficient code, like the following loop that multiplies two input vectors containing floating point values, and stores the result in an output vector `res`:

```
int map_mul_flt_col_flt_col(int n, flt* res, flt* col1, flt* col2, int *sel)
{
   for(int i=0; i<n; i++)
      res[sel[i]] = col1[sel[i]] * col2[sel[i]];
   return n;
}
```

Here, `sel` is a *selection vector* that contains the pivots of those input tuples that match some (earlier evaluated) selection predicate, avoiding the need for expensive vector compacting. For each tuple, this code requires three load instructions, a multiplication, and a write, for a total of 5 instructions. Combining this with a measured average time of 2.1 cycles per tuple, as found in Table 4.2, we can conclude that this primitive achieves a decent IPC of 2.4. As a comparison, MySQL consumes an average of 49 cycles per multiplication, almost a factor hundred worse.

input count	time (us)	avg. cycles	**Vectorwise primitive**
6M	13307	3.0	`select_lt_usht_col_usht_val`
5.9M	10039	2.3	`map_sub_flt_val_flt_col`
5.9M	9385	2.2	`map_mul_flt_col_flt_col`
5.9M	9248	2.1	`map_mul_flt_col_flt_col`
5.9M	10254	2.4	`map_add_flt_val_flt_col`
5.9M	13052	3.0	`map_uidx_uchr_col`
5.9M	14712	3.4	`map_directgrp_uidx_col_uchr_col`
5.9M	28058	6.5	`aggr_sum_flt_col_uidx_col`
5.9M	28598	6.6	`aggr_sum_flt_col_uidx_col`
5.9M	27243	6.3	`aggr_sum_flt_col_uidx_col`
5.9M	26603	6.1	`aggr_sum_flt_col_uidx_col`
5.9M	27404	6.3	`aggr_sum_flt_col_uidx_col`
5.9M	18738	4.3	`aggr_count_uidx_col`

Table 4.2: Vectorwise performance trace during TPC-H Query 1 (primitives).

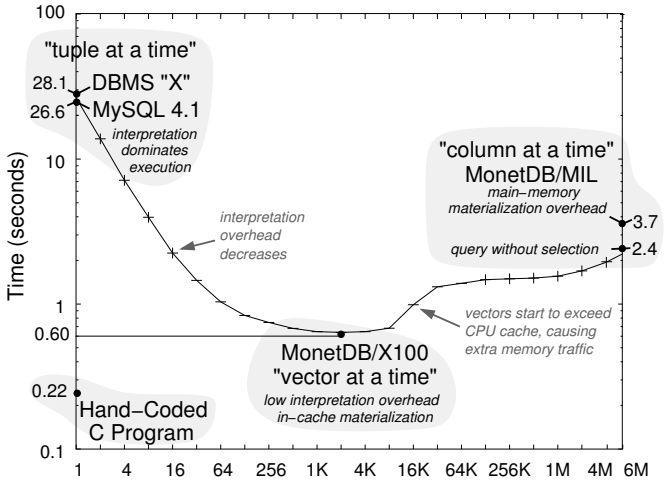


Figure 4.3: Vectorwise TPC-H Q1 performance as a function of vector size.

Figure 4.3 shows the impact of *vector size* on the performance of TPC-H Query 1. In this experiment, the vector size is varied from 1, emulating tuple-at-a-time execution, to six million (the size of the lineitem table), emulating a column-at-a-time execution model. The results are in line with our timings from Table 4.1. At vector size one, the performance of Vectorwise is comparable to the traditional DBMSs, with interpretation overhead in the iterator pipeline being the main bottleneck. As the vector size is increased, performance improves by two orders of magnitude, reaching a "sweet spot" between 1-4K. Further increases in vector size hurt the performance, as vectors grow too large to fit the cache, resulting in an increase in the number of cache misses. At a vector size of six million, the performance approaches that of the column-at-a-time execution of MonetDB/MIL, with each primitive materializing its full output. The 1.3 second difference is due to the fact that Vectorwise uses selection vectors, thereby avoiding full materialization within the select operator, while MonetDB/MIL simply materializes the 99% selected tuples.

For simple sequential computations, column vectors in DSM provide optimal

performance. However, for algorithms that require random lookups, like hashing, it can pay off to convert rapidly back and forth between DSM and NSM tuple layouts, as NSM is more friendly to random access patterns. In [ZNB08], it is shown that converting from DSM to NSM right before the final aggregation in query 1, its performance can be improved slightly, reducing execution time by 9%. In [HNZB07] we show further performance improvements by mapping the vectorized query execution model to the heterogeneous multi-core architecture of the Cell Broadband Engine [IBM07].

4.4 Storage Management

The vectorized execution engine of Vectorwise is capable of achieving excellent performance in main memory scenarios. However, many analytic workloads analyze data volumes that do not fit in main memory, often relying heavily on full table scans or range scans, which are sequential in nature. Scaling Vectorwise's in-memory performance to disk based secondary storage is a significant challenge, as queries can easily consume data at rates of gigabytes per second. To be able to sustain such throughput rates using commodity RAID systems, Vectorwise's storage and buffer manager, ColumnBM, uses three techniques: columnar storage (DSM), compression and intelligent buffer management.

4.4.1 DSM

To optimize for sequential scans, ColumnBM stores relations in columnar fashion, one attribute per file, a storage format also known as decomposition storage model (DSM) [CK85]. Such a layout saves disk bandwidth when queries scan only a subset of table attributes. Furthermore, a sequential array organization in RAM is efficient in terms of memory bandwidth and cache utilization, and integrates nicely with vectorized execution, enabling compilers to produce CPU efficient code.

Due to these benefits, since 2005, column-stores have gained enormous momentum, both in research and industry. This was the year in which both MonetDB/X100 [BZN05] and Stonebreaker's' C-store [Sto05], an open-source research column-store, were introduced. Earlier, the only relevant column-stores were the open-source MonetDB [Mon], X100's predecessor, and the commercial Sybase-IQ [Syb]. Both X100 and C-store have since then been commercialized, resulting in Vectorwise [Act] and Vertica [Ver] respectively, with Stonebreaker still claiming column-stores to be the future for analytic DBMSs [SMA+07, SR13].

4.4.2 Compression

To further improve I/O performance of individual queries, Vectorwise employs *ultra lightweight compression* and *into-cache decompression* [ZHNB06] techniques that aim at increasing the data output rate of scan operators by reading compressed data from disk and decompressing it on-demand, in a vectorized fashion, directly into the operator pipeline, as illustrated in Figure 4.2. The main goal being a speedup in query execution, even on RAID systems, rather

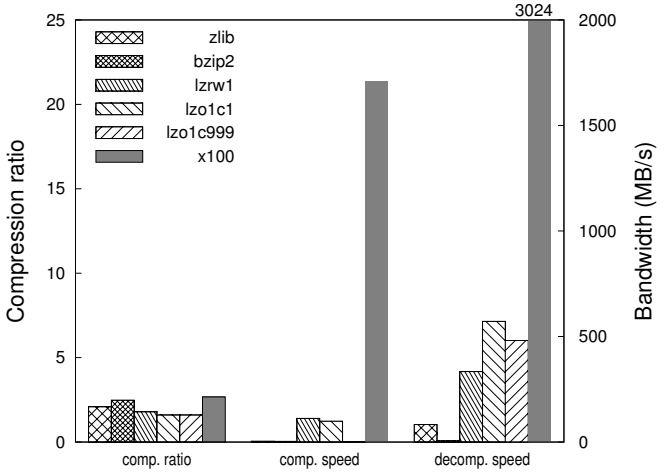


Figure 4.4: Performance of various compression algorithms on TPC-H extended_price column.

than maximization of the compression ratio, the traditional goal of data compression. This focus on decompression speed sets compression in Vectorwise apart from earlier research into database compression [GS91, RvH93].

However, a focus on speed does not imply that compression ratios can not be competitive, as shown in Figure 4.4, where we see both compression ratio and (de)compression speeds of Vectorwise being compared against traditional algorithms (bzip2 [BW94], zlib [ZL77], and its speed optimized derivatives LZRW [Wil91] and LZO [Mar]) on a single column from TPC-H. By compressing on a per-column basis, Vectorwise can exploit domain and type specific knowledge of each attribute to combine fast (de)compression with good compression ratios, using algorithms that can be implemented as simple, CPU-efficient loops over column-vectors. Figure 4.4 confirms that Vectorwise is able to deliver superior (de)compression speeds in the order of gigabytes per second, without compromising compression ratio. Such speeds allow Vectorwise to decompress data at a faster rate than typical commodity RAID systems can deliver, allowing for speedup of I/O-bound queries that is comparable to the compression ratio, as illustrated in Figure 4.5.

Most database systems employ decompression right after reading data from disk, storing buffer pages in uncompressed form. This requires data to cross the boundary between memory and CPU-cache three times: for decompression in the CPU, writing the uncompressed data back to memory, and finally when it is read by a query. Since such approach would make Vectorwise decompression routines memory-bound, ColumnBM stores disk pages in a compressed form and decompresses them just before execution, on a per-vector granularity. Thus (de)compression is performed on the boundary between CPU cache and main memory, rather than between main memory and disk.

Compression in Vectorwise is discussed in detail in Chapter 5, with applications in the field of information retrieval in Appendix A, where we use it to compress inverted files. Under certain conditions, decompression is not even needed at all, as database operators can be extended to operate on compressed

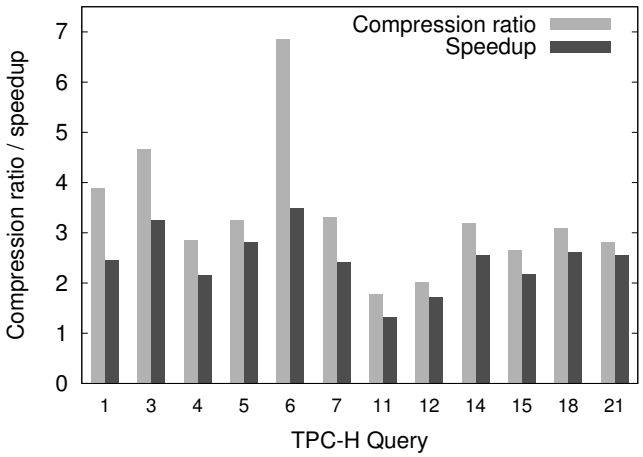


Figure 4.5: Compression ratio and speedup on a subset of TPC-H queries.

data directly. This is investigated in [Lus11].

4.4.3 Buffer Management

To avoid waste of precious disk bandwidth, ColumnBM employs buffer manager (BM) policies that aim to maximize sharing of cached I/O pages among queries that concurrently scan the same columns. Traditionally, database systems have used buffer manager policies like LRU or MRU, trying to optimize for access patterns that are prevalent in point-queries. In scenarios with mostly concurrent scans, however, these policies perform poorly, as scans end up competing for disk access. Besides increasing the latency of individual queries, which take turns in accessing the I/O subsystem, such policies also decrease overall I/O throughput, as competing scans break locality of access, which may lead to thrashing.

To remedy this situation, several database systems incorporated use of *circular scans* [Coo01], where incoming queries attach to an already ongoing scan, limiting thrashing. By *throttling* fast scans to the speed of an overlapping but slower scan, sharing opportunities, and thus overall throughput, can be further improved [LBM+07]. Within the context of Vectorwise, two distinct approaches to scan sharing have been investigated: *cooperative scans*, or CScans, and *predictive buffer management* (PBM). We briefly discuss both of them below.

Cooperative Scans

Cooperative scans [ZHNB07] rely on a global *active buffer manager* (ABM) component, with which each scan has to register its page demands before it starts execution. This gives ABM an overview of all scans that are running, together with their I/O demands. Using this knowledge, ABM can adaptively decide which pages to load and to which scans these pages should be passed, trying to keep as many scans busy as possible. The major consequence of this is that individual scans will typically receive pages in a seemingly arbitrary order that does not correspond to the physical table layout. We label such out-of-order scans `CScan`, to differentiate them from traditional scans.

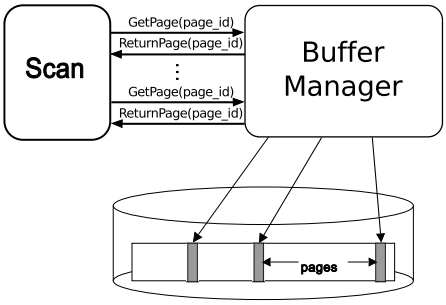


Figure 4.6: Traditional BM.

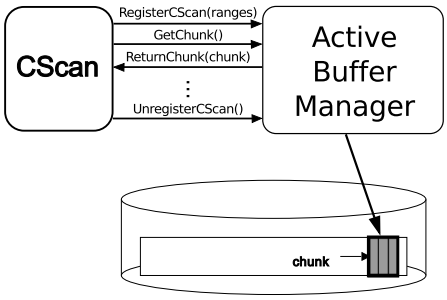


Figure 4.7: Cooperative Scans.

Figure 4.7 contrasts ABM's behavior with traditional buffer management, Figure 4.6, where individual scans govern the order of page requests. The traditional buffer manager always returns the requested page, and also acts as a cache, trying to keep relevant pages buffered using an eviction algorithm like LRU or MRU. In ABM, on the other hand, after registering its I/O needs through `RegisterCScan(ranges)`, each `CScan` performs an unparameterized `getChunk()` request, which returns a relatively large, sequential range of tuples (i.e. "chunk") that still needs to be processed by the requesting scan. To decide which `CScan` to service, and which chunks to load, evict and return ABM makes use of four *relevance functions*, which aim to optimize both the latency and throughput of queries.

In short, `QueryRelevance()` is computed over all active `CScan`s to decide which one most urgently needs data. It tries to prioritize *starved* queries, which have (almost) none of their remaining data available in the buffer pool. After a `CScan` has been picked, `LoadRelevance()` is computed on its remaining chunks, prioritizing chunks that many other `CScan`s are interested in, thereby maximizing buffer reuse. In case a `CScan` has multiple chunks available in memory, ABM computes `useRelevance()` on those to decide which one to return to the scan. This process prioritized chunks with the fewest number of interested `CScan`s, thereby making them available for eviction as soon as possible. These are also the chunks that score low on `KeepRelevance()`, and are therefore selected as the chunk to evict in case we have to free up buffer space.

To shed some light on the benefits that active buffer management could bring, Figure 4.8 shows TPC-H SF30 throughput numbers for a varying number of concurrent streams, each of which runs all the 22 read-only queries from the benchmark (no update streams). In the top figure, we see that for an increasing number of streams, average stream time goes up, as streams are competing for resources. However, timings for the `CScan` policy deteriorate less rapidly than for a traditional LRU policy, especially in highly concurrent scenarios, where the sharing potential is larger. The bottom figure shows the benefits in terms of I/O volume reduction, which is the main cause of the improved execution times.

Although `CScan`s can provide a significant benefit, it has remained a research project in Vectorwise. During efforts to integrate `CScan`s into the industrial Vectorwise system, a few practical complexities were found, most notably in the areas of parallelism, transactional updates, and support for coexistence with regular (i.e. sequential) scans, something that is desirable to support efficient operators that rely on physically ordered data. While investigating solutions to

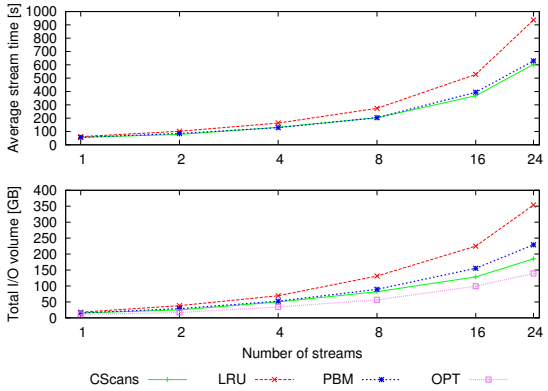


Figure 4.8: TPC-H throughput varying the number of streams.

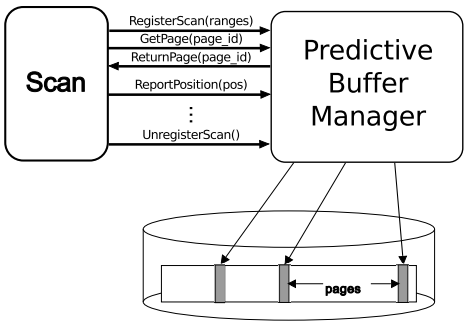


Figure 4.9: Predictive Buffer Manager.

these problems, an alternative was found in *predictive buffer management*.

Predictive Buffer Management

Predictive buffer management (PBM) [ŚBZ12, Ś11], borrows from a well known and proven optimal algorithm for cache replacement, OPT [Bel66]. In OPT, when a page needs to be evicted, the system selects the page whose next use will occur farthest in the future. Clearly, this algorithm has limited practical use, as it relies on full knowledge about the order of future page requests, knowledge that is generally not available on systems with ad-hoc workloads. However, in an analytic DBMS, with long running scans, predictions about future page requests can be made quite reliably. PBM exploits this observation to estimate the *time of next consumption* for pages that are subject of a table scan, by monitoring the speed at which each scan progresses through the physical data.

Like `CScan`s, predictive buffer management (PBM) relies on knowledge about currently running queries. In contrast, it does not rely on a global object where all decisions regarding loading and evicting pages are made. Similar to traditional scan handling, PBM lets each scan initiate I/O requests and does *not* change the order of those requests, as can be seen by comparing Figure 4.9 to Figure 4.6. This allows PBM to be easily integrated into an existing DBMS, without a need to change operator implementations or physical storage organi-

zation. As we can see in the TPC-H results from Figure 4.8, PBM is able to perform comparably to `CScan` in terms of average stream time, even though it scans a larger volume of data compared to `CScan` and the ideal OPT [ŚBZ12]. Due to its considerably lower software complexity compared to `CScan`s, the slightly worse performance of PBM over the line was taken for granted, making PBM the buffer manager of choice in Vectorwise.

4.4.4 Indexing

To avoid scanning irrelevant data, scans should target the physical regions on disk that contain data relevant to a query. This calls for some form of indexing. Traditional database indexing of tuples at disk page granularity does not work well for a compressed column-store like Vectorwise. Vertical partitioning of tuples into columns implies that a page on disk stores values from a single attribute only. As long as the index is on a single attribute, page-level indexing could still work. However, it becomes complicated as either the number of indexed attributes or the number of attribute values retrieved by a query grows beyond that single indexed attribute. The former scenario requires maintenance of multiple page pointers (i.e. one per indexed attribute), while the latter calls for an easy and efficient way to reconstruct any randomly picked tuple.

To avoid these complications, Vectorwise employs *positional indexing*, which allows for fast positional tuple reconstruction. *Value-based indexing*, which can be used to reduce scan volume by pushing down selections, is made available on top of this by means of a *MinMax index*, which provides summary information for logical horizontal partitions (i.e. position ranges) of a table. Furthermore, Vectorwise allows propagation of positional ranges between two tables that participate in a foreign-key constraint through a join index. These indexing strategies are discussed in the following sections.

Position Index

At the lowest level, Vectorwise employs a page-level *position index*, as depicted in Figure 4.10, where we see the storage layout of a three attribute table representing bank accounts with their holders and balances. Each page holds data from a single attribute, often in compressed form. Because of such compression, and variable width data types in general, each densely packed page contains a varying number of values. I.e. the first page with account numbers contains eight values, while the first page of the name column only holds three. This means that we cannot easily reconstruct an arbitrary tuple, which requires us to fetch the i'th attribute value from each column to reconstruct the i'th tuple of the table. To solve this, we maintain a position index for every storage column, which is a sorted, in-memory array containing the positional offset of the first attribute value in each page, together with a (logical) pointer to the actual page data. Using binary search on this position array, we can quickly pinpoint the pages containing the relevant attribute values for each attribute of a tuple.

Both the densely packed compressed storage and the position index are structures that are costly to update. Therefore, we consider those storage structures immutable, never updating them in-place, as discussed further in Chapter 7. This immutable version of a database we call its *stable image*, and the fixed position (i.e. offset) of each tuple *stable ID* (SID).

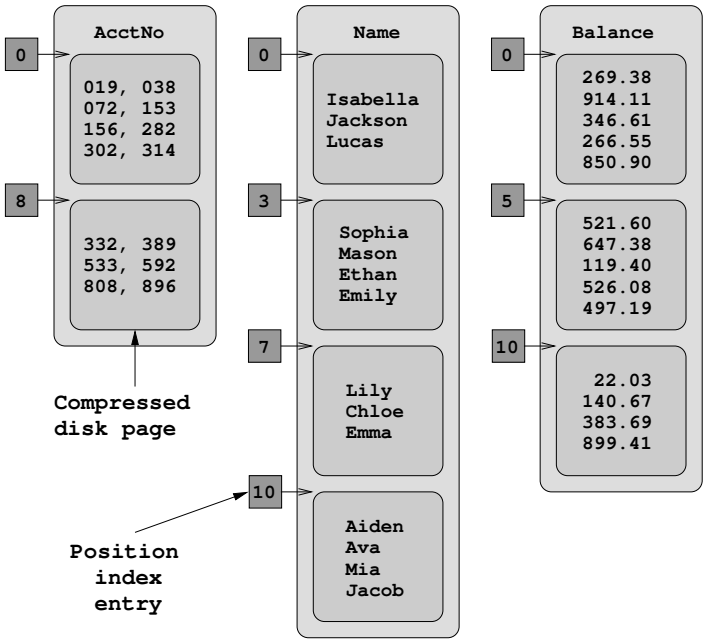


Figure 4.10: Positional page-level indexing.

Other indices in Vectorwise use such SID positions, rather than page pointers, as a way to refer to horizontal ranges of vertically decomposed tuples. I.e. each index returns one or more SIDs, which can then be mapped to the relevant pages through the position index, resulting in a two-level indexing architecture that allows us to reconstruct (partial) tuples relatively efficiently.

The position index is also used heavily during parallel query execution. Each scan that gets parallelized results in several disjoint horizontal partitions, or positional tuple ranges, which need to be mapped to physical pages using the position index.

MinMax index

In an effort to reduce I/O during scan-select operations, Vectorwise correlates positional indexing with actual attribute values in a *min-max index*. To achieve this, the min-max index logically partitions a table into M buckets, each with an equal number of consecutive tuples. A special overflow bucket at the tail can hold an arbitrary remainder of tuples. The value of M is a constant, and is determined at database creation time, so larger tables contain more tuples per partition. For each partition (or *bucket*), the index maintains the starting position (SID) and minimum and maximum values for all or select attributes. Figure 4.11 shows an example partitioning (for $M = 4$) of our example `accounts` table, together with starting positions and min-max values for each attribute within the resulting buckets. In practice, the value for M is greater than four, but still small, so that the entire min-max index easily fits in RAM. A typical value of $M = 1024$ divides the table into buckets containing roughly 0.1% of the data, with the index usually consuming less than a megabyte.

Accounts			
SID	**acctno**	**name**	**balance**
00	019	Isabella	269.38
01	038	Jackson	914.11
02	072	Lucas	346.61
03	153	Sophia	266.55
04	156	Mason	850.90
05	282	Ethan	521.60
06	302	Emily	647.38
07	314	Lily	119.40
08	332	Chloe	526.08
09	389	Emma	497.19
10	533	Aiden	22.03
11	592	Ava	140.67
12	808	Mia	383.69
13	896	Jacob	899.41

Accounts.MinMax							
bucket	**SID**	**acctno**		**name**		**balance**	
		min	**max**	**min**	**max**	**min**	**max**
0	00	019	153	Isabella	Sophia	266.55	914.11
1	04	156	314	Emily	Mason	119.40	850.90
2	08	332	592	Aiden	Emma	22.03	526.08
3	12	808	896	Mia	Jacob	383.69	899.41

Figure 4.11: Example MinMax index for `accounts` table.

To illustrate the usage of the min-max index, consider the following scan-select query:

```
SELECT * FROM accounts WHERE name > 'Lara' AND balance < 200
```

From the selection on `name`, we know that all buckets except bucket 2 contain relevant data. This would allow us to restrict the scan to the union of tuples in the ranges [0, 8) and [12, 14) (note that upper bounds are exclusive). We can, however, further reduce the ranges by intersecting with the buckets matched by the selection on `balance`, which matches both buckets 1 and 2. Bucket 1 is the only bucket that satisfies both clauses, meaning that we can restrict our scan to SIDs [4, 8). This final range can be mapped to the page-level position index, as found in Figure 4.10, where we find that for the `name` attribute we need to scan two out of four pages (the second and third one), and for `balance` only one out of three pages (the second one). In this example, the min-max index allows us to reduce scanned I/O volume by 57% compared to a full scan of both columns.

A min-max index is most effective on (nearly) sorted columns, as the range of relevant tuples will be highly localized. An imperfect ordering may occur for attributes that are correlated with the main sort attribute of a table, for example a time stamp field that is correlated with an automatically incremented tuple ID. In general it holds that the more uniform the distribution of attribute values, the less useful min-max becomes.

Table P

SID	SK	JK
0	a	M
1	c	D
2	e	X
3	g	O
4	j	S

JoinIndex

SID_P	SID_C
0	0
0	1
2	2
2	3
2	4
3	5
4	6
4	7

Table C

FK	*SID*
M	0
M	1
X	2
X	3
X	4
O	5
S	6
S	7

Figure 4.12: A (clustered) join index between parent table P and child table C.

Join index

To propagate scan ranges, as introduced by selection push-down or range partitioning for parallelism, over joins, Vectorwise supports a simplified form of join index. The original join index [Val87], as summarized in Section 3.3.3, correlates matching tuples in a parent and child table by storing a pair of unique integer tuple IDs, one for the parent and one for the child. Efficient lookups and updatability are then provided by means of an index on those tuple IDs (or *surrogates*), which are assumed to be physically stored and indexed in both the index and the base relations.

Given the static nature of Vectorwise's column storage and its positional indexing, we adjust the join index to this design philosophy. We do this by correlating tuple offsets (SIDs) rather than surrogates, as these do not need to be stored in the base relations. This layout is illustrated in Figure 4.12, where we see two tables, a *parent*, or *referenced*, table, `P`, and a *child*, or *referencing*, table, `C`. Table `P` is stored sorted on *sort-key* attribute `SK`, and has a second attribute, `JK` ("join key") which is being referenced by the foreign key, `C.FK`, in the child table. Both tables could contain more attributes, but this is a minimal example. Note that the SID column of each table is solely there to depict the enumerated tuple positions. These are not stored physically on disk. The join index, which correlates matching tuples by means of SID, does get stored though. The join index contains for given tuples in `C`, at position `SID_C`, the position of the matching parent tuple, `SID_P`.

The example in Figure 4.12 shows a *clustered* join index, where the child table is sorted according to the sort order of `P`, i.e. sorted on `P.SK` after performing the foreign key join. In the general case, the `SID_C` offsets may be scattered all over the child table, resulting in a more random access pattern. Vectorwise only supports clustered join indices, as the cost of random tuple lookups rapidly outweighs the cost of a sequential scan, especially in a column-store. Therefore, the `SID_C` column always contains a densely increasing sequence (a natural ordering) of the tuples in `C`, which can safely be removed, given its immutable nature. Furthermore, the remaining `SID_P` column is a good candidate for

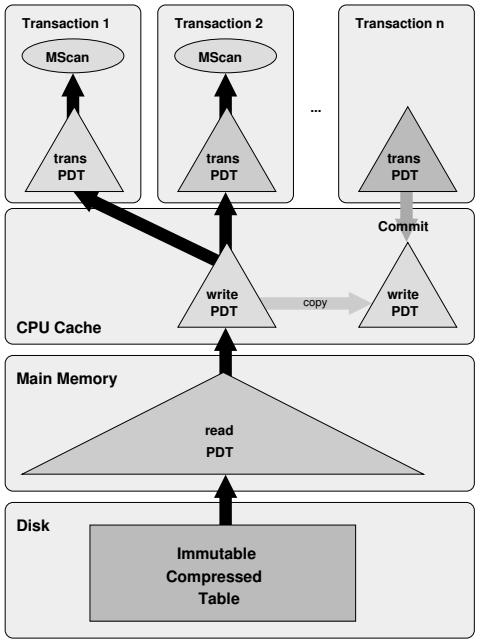


Figure 4.13: Stacked layers of PDTs containing positional differential updates.

compression, a topic that we leave for Section 7.4, where we discuss how to maintain this, rather static, join index representation under updates.

Finally, join indices may be used to connect more than two tables into so called *cluster trees*, where tables are nodes and the join index relationships make up the edges. In a cluster tree, any parent table can have multiple child tables associated with it. Each child table, in turn, can be the parent of next level of child tables in a recursive tree structure. Such cluster trees can be used to replace expensive multi-way hash join based plans with their faster, merge-based counterparts.

4.5 Transactional Updates

Columnar storage using densely packed, compressed disk pages is rather unfriendly towards updates. To avoid direct manipulation of disk pages, Vectorwise maintains differential updates [SL76] for each table in a memory resident data structure called *positional delta tree* (PDT) [HZN+10]. These updates are then merged into a scan on-the-fly to provide an up-to-date table image. To optimize this merging, PDTs organize updates by *tuple position* (i.e. offset) rather than some key of attribute values. This allows updates to be applied using only integer arithmetic and avoids scanning of key attributes solely for the purpose of update merging. The net result is that Vectorwise provides support for updates with a minimal impact on its excellent scan performance.

An overview of the PDT architecture is provided in Figure 4.13. Here we see three layers of PDTs, *read-*, *write-* and *trans-PDT*. The read-PDT contains differential updates with respect to an immutable table on disk. The write-PDT, in turn, contains differential updates against the same table, but *with updates*

from the read-PDT applied, akin to the *log-structured merge tree* (LSM-tree) [OCGO96].

Each trans-PDT is private to a transaction, providing isolation, and contains all transaction local changes with respect to the table image with read- and write-PDT updates included. The `MScan` ("merge scan") operator applies updates from all PDT layers to produce a current view of the table. When a transaction commits, its trans-PDT changes are propagated into a *snapshot copy* of the global write-PDT, allowing ongoing scans to keep using the old write-PDT. The write-PDT is kept small, i.e. cache sized, to *(i)* avoid cache misses, and *(ii)* avoid expensive snapshot copies of the larger read-PDT. Updates from this large read-PDT are periodically merged into a new version of the immutable table on disk, allowing read-PDT memory to be freed.

Full details about the PDT data structure and its application to transaction processing are provided in Chapter 6. In Chapter 7 we discuss index maintenance, including maintenance of Vectorwise's positional join indices, which pose significant challenges.

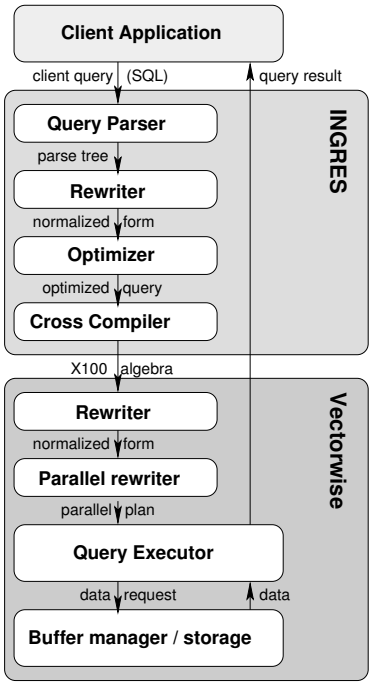
4.6 Vectorwise

In august 2008 a commercial agreement was made between CWI and Ingres Corp. (now Actian) to integrate Vectorwise technology with the Ingres database server, the goal being to produce a viable commercial RDBMS that provides the accepted components of enterprise class database management, together with the extreme performance of the academic MonetDB/X100 engine [IZB11]. The 1.0 version of Vectorwise was released in June 2010. In 2011 official results were published on the TPC-H benchmark [Tra02] for scale factors 100GB, 300GB and 1000GB, claiming the top spot on the first two, using a single server machine. We briefly summarize the integrated system architecture and some of the research efforts below.

4.6.1 SQL Front-end

Vectorwise reuses most of the SQL front-end from Ingres, as the Vectorwise back-end (the original X100 server), lacks SQL support. The front-end and back-end are organized in a client-server architecture, running in separate processes, communicating through a client-server protocol with support for DDL, DML and queries to be submitted, for session and transaction management, and for result rows to be returned to the SQL client. An overview of the system architecture can be found in Figure 4.14a. Details about the integration of X100 into Ingres can be found in [IZB11, ZvdWB12].

The query interface to the back-end consists of a textual representation of low-level relational algebra, where plans are written as algebraic trees built from operators like `Scan`, `Select`, `Project`, `Aggr` and `CartProd`. A simple example translation can be found in Figure 4.14b. Submitted plans are expected to be "hand optimized", as no major plan optimization is performed within the back-end itself. Therefore, Vectorwise plans specify physical operator implementations. For example, an `OrdAggr` ordered aggregation can be chosen over the default hash based `Aggr` in case the inputs are known to be sorted. Similarly, joins can select between `HashJoin` and `MergeJoin`, or one of their more



(a) Vectorwise Architecture.

```
SQL:
SELECT a, sum(b) AS c FROM T
WHERE a > 10
GROUP BY a;
```

```
Project(
   Aggr(
      Select(
         MScan('T', ['a', 'b']),
         [a > 10]
      ),
      [T.a], [c = sum(T.b)]
   ), [a, c]
)
```

(b) Query interface.

Figure 4.14: Vectorwise client-server architecture and query interfaces.

specific variants (like outer joins).

The Vectorwise back-end has no support for traditional nested loop joins nor for sub-queries. It does, however, support *re-use* of operator outputs, i.e. results from an operator may, under certain conditions, be fed into a higher level operator within the same query plan to avoid needless re-computation of common expressions. This allows Vectorwise plans to be proper graph shaped rather than strictly tree shaped.

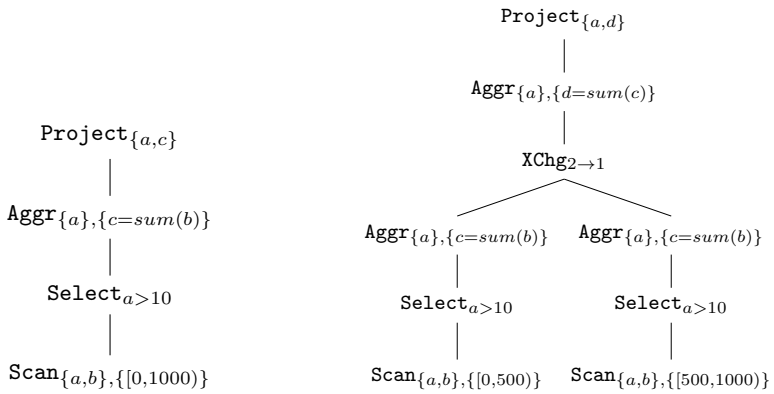
The client reuses large parts of the regular Ingres SQL client, like the parser and a modified optimizer [IZB11, ZvdWB12]. The Ingres optimizer is cost based and relies on table statistics, like histograms, to be present. Such statistics are extracted from Vectorwise tables and maintained in a client-side metadata schema used by the optimizer. The optimizer uses those to perform traditional rewrites, like join reordering and selection push-down.

The front-end also incorporates several Vectorwise specific enhancements. First of all, SQL subqueries are *flattened* by the Ingres rewriter [IZB11], as the back-end does not support them. Second, the optimizer is now aware of foreign key constraints, and, more specifically, of join index presence, which allows highly efficient `MergeJoin`s, based on tuple position, to be used. Third, the Ingres optimizer is extended with a concept of clustering, which is weaker than ordering, but still allows for operators like `OrdAggr` to be used. And finally, making the optimizer fully column-store aware is perhaps the biggest challenge within the optimizer, and is still a work in progress.

Another component with many Vectorwise specific changes is the query com-

piler, which was changed into a Vectorwise *cross compiler*, responsible for producing Vectorwise algebra plans rather than physical Ingres query execution plans. The cross compiler takes an optimized Ingres plan as input, and outputs textual Vectorwise algebra that can be readily submitted to the back-end.

The back-end converts the query plan into an execution tree as depicted in Figure 4.15a, which is executed by the vectorized query engine, returning its results to the SQL client. The back-end does not perform cost based plan optimization. There are, however, some rewrites and other optimizations performed by a rule-based query *rewriter*. For example, the rewriter uses information from the min-max indices to annotate scans with offset *ranges*. Also, scans involving nullable columns are extended to scan along a special Boolean *null column*, which indicates whether the value in the corresponding *value column* is valid or should be null. Proper handling and propagation of such null columns is also handled by the rewriter, in such a way that null checks in core computational loops (vectorized primitives) are avoided. Finally, the rewriter is responsible for mapping logical types to physical types, and for two optimizations: dead code elimination (DCE) and parallelization of query plans.



(a) Sequential plan. (b) Parallel plan.

Figure 4.15: Example transformation of a sequential plan (a) into a parallel plan (b).

4.6.2 Parallelism

Vectorwise supports traditional intra-query parallelism using Exchange (`XChg`) operators [Gra90, Ani10] that fit nicely into a Volcano iterator pipeline. Parallelism is (optionally) introduced in the Vectorwise back-end by transforming a query plan in such a way that it contains multiple copies of certain subtrees and connecting those using a special `XChg` operator. The subtrees are then executed in parallel by multiple threads. An example of such a transformation can be found in Figure 4.15.

The subtree with `Aggr` as a root is duplicated and connected using an `XChg` operator that merges the two result streams into one. Given that aggregation is a global operation, a second `Aggr` is introduced that combines the aggregated results from both subtrees into an overall result. The main benefit stems from

the fact that each parallel `Scan` now only reads (and outputs) half the tuple range scanned by the sole `Scan` in the sequential plan. These partitioned halves can then be processed independently and in parallel within the two subtrees. In general, each `Scan` operation is range partitioned into P independent and equally sized tuple ranges, with P being the number of parallel threads.

4.6.3 Research Topics

To investigate further improvements to the vectorized execution model, several research projects have been done within the context of Vectorwise.

Just-in-time compilation

In [SZB11, Som11], *just-in-time (JIT) compilation* of (parts of) queries into machine code is investigated, both for a traditional tuple-at-a-time engine and the vectorized Vectorwise engine. The underlying idea is that compilation removes interpretation overhead and can result in very concise and CPU friendly code. In regular vectorized execution, expressions are built from several primitives, each of which materialize their output in result vectors. Although such input and output vectors are kept in CPU caches as much as possible, compilation could avoid materialization entirely by keeping intermediate results in CPU registers.

In practice, however, evidence in support of query compilation turns out to be marginal. For arithmetic expressions in a `Project` operation, a compiled loop can provide some speedup (less than a factor 2 in [SZB11]). For more complex operators like `Select` and `HashJoin`, results of compilation are less satisfactory. The authors attribute this to *(i)* missed SIMD opportunities, *(ii)* no way to avoid branches in condition evaluation loops (while regular vectorization can use selection vectors) and *(iii)* parallel memory access for the tight loops of independent loads in the vectorized model. Therefore, compilation should not be applied bluntly, only in combination with run-time adaptive techniques. For tuple-at-a-time engines, improvements gained by switching to a vectorized engine (factor 50) far outweigh the benefits of incorporating JIT compilation (factor 3).

Micro Adaptivity

Like JIT compilation, *micro adaptivity* [RBZ13, RŤ12] tries to improve the performance of vectorized primitives, as this is where the largest part of query execution time is spent (typically more than 90%). Primitive efficiency depends not only on the implemented algorithm and the way it got compiled, but also on environmental factors, like the hardware platform, data distributions, query parameters and any concurrently running workloads. The great complexity of modern hardware platforms and compiler techniques, combined with the dynamic influences of the environment make it impossible to select a single optimal implementation for a given set of primitives.

Micro adaptivity aims to address above problem, by allowing multiple implementations, or "flavors", of the same primitive to coexist in the query engine. During execution, on every invocation of a primitive, micro adaptivity aims to select the optimal implementation, based on historical and current performance of available flavors, thereby providing adaptation to different hardware platforms

and run-time changes in environment. Besides improving performance robustness, micro adaptivity also saves development time spent in tuning heuristics and cost modeling thresholds for traditional query optimization. Besides exposing the potentially large performance differences between distinct primitive implementations, the authors of [RBZ13] propose a greedy learning algorithm that is able to beat a setup that makes use of static, hand-tuned heuristics by 9% on average on the TPC-H benchmark.

Recycling results

Data analysis and decision support applications typically generate complex queries that access large volumes of data, are heavy on aggregations, and have relatively small result sizes. Often, these queries are small variations of a few query patterns, especially if they are generated by a reporting or OLAP tool. Such similarity between queries suggests intermediate or final results could be shared, avoiding needless recomputation to improve execution times. Sharing of (intermediate) results in the context of Vectorwise is explored in [NBV13, Nag10], where it is called *recycling*.

Recycling in pipelined query evaluation, where intermediate results are never materialized by default, calls for explicit materialization decisions to be made about which results to cache and for how long, as saving results to memory slows down execution and occupies system memory. Therefore, such a system must perform a run-time cost-benefit analysis for each potential intermediate result before it is actually created and, hence, before its exact cost and size are known.

In [NBV13] a recycler architecture is presented that automatically identifies and exploits reoccurring query patterns by selectively materializing intermediate results, and reuses these to accelerate subsequent queries. It adapts to changes in the workload without need for DBA intervention. A benefit metric is proposed that helps in deciding which results to materialize. The metric is computed using a *recycler graph*, a unification of query trees from previous invocations, annotated with statistics, against which incoming plans can be matched. This graph acts as an index into a *recycler cache* which contains cached intermediate results. The matching algorithm identifies both opportunities for reuse, and for new intermediates to be added to the cache. It is shown that substantial benefits can be obtained on the TPC-H benchmark.

4.7 Related Work

The idea of columnar storage was already proposed back in 1985 by Copeland and Khoshafian as the decomposition storage model [CK85]. Since then, many prototype and commercial column-store systems have been developed. The first widely known commercial column store, launched in 1996, is Sybase IQ [Syb], which was acquired by SAP in 2010.

MonetDB [BMK99, Bon02], under development since the early nineties at the Dutch research institute CWI, was one of the early pioneers, and is still available under an open source license [Mon]. Vectorwise has its roots in MonetDB, but differs in several crucial areas, most notably: it uses vector-at-a-time rather than column-at-a-time execution model and it was designed to scale out

of memory by integrating buffer management and ultra-lightweight compression.

Interest in column stores was rekindled in 2005 by a paper from Stonebreaker, introducing the C-store DBMS [Sto05]. In this same year, the first paper describing a prototype for Vectorwise was published [BZN05]. C-store later evolved into its commercial counterpart, Vertica [Ver, LFV+12], which got acquired by Hewlett Packard. Vectorwise was commercialized by Ingres, which was recently rebranded to Actian, soon after which the product was renamed to Vector [Act].

In this period, a multitude of novel column store systems has been introduced, including players like Exasol [Exa], Infobright [SE09], SAND [SAN], SAP Hana [FML+12] and Paraccel [Par], which is now part of Actian as well. Besides Actian Vector, several other commercial systems support both row-wise and column-wise storage, including Greenplum [Gre], Teradata [Ter], Microsoft SQL server [Mic, LCH+11, LCF+13], and IBM BLU [RAB+13], which is similar to Vector in many respects. Trade-offs between row- and column-wise storage (NSM vs. DSM) have been analyzed in [HLAM06, AMH08]. PAX [ADHS01] stores full rows in each disk block, but changes the data layout within a block to achieve cache behavior as good as DSM.

Block-oriented query processing was introduced in a traditional NSM context in [PMAJ01], with a reduction in the number of function calls as the main goal. Later work showed that it can also be put to use to improve instruction cache hit-ratio [ZR03a, ZR04], or to enable performance enhancing techniques that require multiple tuples to work on, like exploiting SIMD [ZR02] or memory prefetching [CAGM04]. IBM BLU [RAB+13] and SAP Hana [FML+12] are commercial systems known to use vectorized processing.

Buffer management strategies for various disk access patterns were identified in [CD85]. This work and the following [CR93] considered scans trivial, and proposed either LRU or MRU policies to be used. Policies that aimed to improve disk usage under concurrent scan workloads started to be introduced in Teradata [Ter], RedBrick [Fer94] and Microsoft SQL server [Coo01]. These systems either used an elevator algorithm or allowed incoming scans to attach to an already running one. Circular scans, where a single thread repeatedly scans an entire table to simultaneously feed all currently interested queries have been used in Crescendo's "clock scan" [UGA+09] and in QPipe [HSA05]. Non-circular multi-scans are proposed in [LBM+07], where scans can be arranged into groups that are throttled based on scan speed.

Chapter 5

Column Compression

5.1 Introduction

The previous chapter demonstrated the computational power of a vectorized execution engine on memory-resident datasets. For data-hungry queries, this performance easily translates into data consumption rates of one or more gigabytes per second. In terms of main-memory bandwidth this is not a problem. However, in Chapter 2 we learned that read bandwidth from secondary storage is significantly worse (factor 30-80). Only by using high-end SSDs, or by combining multiple slower drives into a RAID setup, can we get in the gigabyte per second range.

To alleviate this I/O bottleneck, we propose novel lightweight data compression techniques that allow us to increase effective disk bandwidth by reading compressed data from disk, and decompressing it at marginal CPU cost right before accessing it. The idea being that by trading CPU power – which goes to waste in an I/O bound scenario anyway – for effective I/O bandwidth, we can improve overall query performance.

5.1.1 Contributions

Our work differs from previous use of compression in databases and information retrieval in the following aspects:

Super-scalar Algorithms

We contribute three new compression schemes (PDICT, PFOR and PFOR-DELTA), that are specifically designed for the super-scalar capabilities of modern CPUs. In particular, these algorithms lack any *if-then-else* constructs in the performance-critical parts of their compression and decompression routines. Also, the absence of dependencies between values being (de)compressed makes them fully *loop-pipelinable* by modern compilers and allows for *out-of-order* execution on modern CPUs that achieve high Instructions Per Cycle (IPC) efficiency. On current hardware, PFOR, PFOR-DELTA and PDICT compress more than a GB/s, and decompress a multitude of that, which makes them more than 10 times faster than previous speed-tuned compression algorithms.

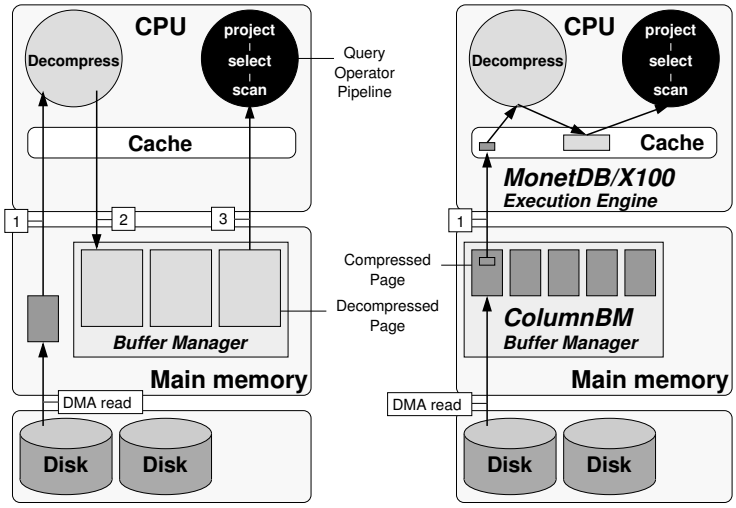


Figure 5.1: I/O-RAM vs RAM-CPU compression.

This allows them to improve I/O bandwidth even on RAID systems that read and write data at rates of hundreds of MB/s.

Improved Compression Ratios

PDICT and PFOR are generalizations of respectively dictionary and Frame-Of-Reference (FOR) or prefix-suppression (PS) compression, that were proposed previously [Ter, GRS98, WKHM00]. In contrast to these schemes, our new compression methods can gracefully handle data distributions with outliers, allowing for a better compression ratio on such data. We believe this makes our algorithms also applicable to information retrieval. In particular, we show that PFOR-DELTA compression ratios on the TREC dataset approach that of a high-speed compression method tuned for inverted files [AM05b] ("carryover-12"), while retaining a 7-fold compression and decompression speed advantage.

RAM-CPU Cache Compression

We make a case for compression/decompression to be used on the boundary between the CPU cache and RAM storage levels. This implies that we also propose to cache pages in the buffer manager (i.e. in RAM) in compressed form. Tuple values are decompressed at a small granularity (such that they fit the CPU cache) just-in-time, when the query processor needs them.

Previous systems [Syb] use compression between the RAM and I/O storage levels, such that the buffer manager caches decompressed disk pages. Not only does this mean that the buffer manager can cache less data (causing more I/O), but it also leads the CPU to move data three times in and out of the CPU cache during query processing. This is illustrated by the left side of Figure 5.1: first the buffer manager needs to bring each recently read disk block from RAM to the CPU for decompression, then it moves it back in uncompressed form to a buffer page in RAM, only to move the data a third time back into the CPU cache, when it is actually needed by the query. As buffer manager pages are

compressed, a crucial feature of all our new compression schemes is fine-grained decompression, which avoids full page decompression when only a single value is accessed.

We implemented PDICT, PFOR and PFOR-DELTA in the ColumnBM storage manager of Vectorwise. Our experiments show that on the 100GB TPC-H benchmark, our compression methods can improve performance with the compression ratio in I/O constrained systems, and eliminate I/O as the dominant cost factor in most cases. We tested our compression methods both using DSM column-wise table storage [CK85] as well as a PAX layout, where data within a single disk page is stored in a vertically decomposed fashion [ADHS01]. While the TPC-H scenario favors the column-wise approach, PAX storage also strongly benefits from our compression, extending its use to scenarios where the query mix contains more OLTP-like queries.

5.1.2 Outline

In Section 5.2 we relate our algorithms to previous work on database compression. Section 5.3 then introduces our new PFOR, PFOR-DELTA and PDICT compression algorithms. We use CPU performance counters on three different hardware architectures to show in detail how and why these algorithms achieve multi GB/s (de)compression speeds. We evaluate the effectiveness of our techniques using Vectorwise on TPC-H in Section 5.4, as well as on information retrieval datasets from TREC and INEX in Section 5.5. We conclude and outline future work in Section 5.6.

5.2 Related Work

Previous work on compression in database systems coincides with our goal to save I/O, which requires *lightweight* methods (compared with compression that minimizes storage size), such that decompression bandwidth clearly outruns I/O bandwidth, and CPU-bound queries do not suffer too great a setback by additional decompression cost. In the following, we describe a number of previously proposed database compression schemes [WKHM00, GS91, GRS98]:

Prefix Suppression (PS) compresses by eliminating common prefixes in data values. This is often done in the special case of zero prefixes for numeric data types. Thus, PS can be used for numeric data if actual values tend to be significantly smaller than the largest value of the type domain (e.g. prices that are stored in large decimals).

Frame Of Reference (FOR), keeps for each disk block the minimum min_C value for the numeric column C, and then stores all column values $c[i]$ as $c[i] - min_C$ in an integer of only $\lceil log_2(max_C - min_C + 1) \rceil$ bits. FOR is efficient for storing clustered data (e.g. dates in a data warehouse) as well as for compressing node pointers in B-tree indices. FOR resembles PS if $min_C = 0$, though the difference is that PS is a variable-bitwidth encoding, while FOR encodes all values in a page with the same amount of bits.

Dictionary Compression, also called "enumerated storage" [Bon02], exploits value distributions that only use a subset of the full domain, and replaces each occurring value by an integer code chosen from a dense range. For example, if gender information is stored in a `VARCHAR` and only takes two values, the column

can be stored with 1-bit integers (`0="MALE"`, `1="FEMALE"`). A disadvantage of this method is that new value inserts may enlarge the subset of used values to the point that an extra bit is required for the integer codes, triggering recompression of all previously stored values.

Several commercial database systems use compression; especially prefix compression of node pointers in B-trees is quite prevalent (e.g. in DB2). Teradata's Multi-Valued Compression [Ter] uses dictionary compression for entire columns, where the DBA has the task of providing the dictionary. Values not in the dictionary are encoded with a reserved *exception value*, and are stored elsewhere in the tuple. Oracle also uses dictionary compression, but on the granularity of the disk block [PP03]. By using a separate dictionary for each disk block, the overflow-on-insert problem is easy to handle (at the price of additional storage size).

The use of compressed column-wise relations in our Vectorwise system strongly resembles the Sybase IQ product [Syb]. Sybase IQ stores each column in a separate set of pages, and each of these pages may be compressed using a variety of schemes, including dictionary compression, prefix suppression and LZRW1 [Wil91]. LZRW1 is a fast version of common LZW [Wel84] Lempel-Ziv compression, that uses a hash table *without* collision list to make value lookup during compression and decompression simpler (typically achieving a reduced compression ratio when compared to LZW). While faster than the common Lempel-Ziv compression utilities (e.g. gzip), we show in Section 5.3 that LZRW1 is still an order of magnitude slower than our new compression schemes. Another major difference with our approach is that the buffer manager of Sybase IQ caches decompressed pages. This is unavoidable for compression algorithms like LZRW1, that do not allow for fine-grained decompression of values. Page-wise decompression fully hides compression on disk from the query execution engine, at the expense of additional traffic between RAM and CPU cache (as depicted in Figure 5.1).

An interesting research direction is to adaptively determine the data compression strategy during query optimization [CGK01, GS91, WKHM00]. An example execution strategy that optimizes query processing by exploiting compression may arise in queries that select on a dictionary-compressed column. Here, decompression may be skipped if the query performs the selection directly on the integer code (e.g. on `gender=1` instead of `gender="FEMALE"`), which both needs less I/O and uses a less CPU-intensive predicate. Another opportunity for optimization arises when (arithmetic) operations are executed on a dictionary compressed column. In that case, it is sometimes possible to execute the operation only on the dictionary, and leave the column values unchanged [Syb] (called "enumeration views" in [Bon02]). Optimization strategies for compressed data are described in [CGK01], where the authors assume page-level decompression, but discuss the possibility to keep the compressed representation of the column values in a page in case a query just copies an input column unchanged into a result table (unnecessary decompression and subsequent compression can then be avoided).

Finally, compression to reduce I/O has received significant attention in the information retrieval community, in particular for compressing inverted lists [WMB99]. Inverted lists contain all positions where a term occurs in a document (collection), always yielding a monotonically increasing integer sequence. It is therefore effective to compress the *gaps* rather than the term posi-

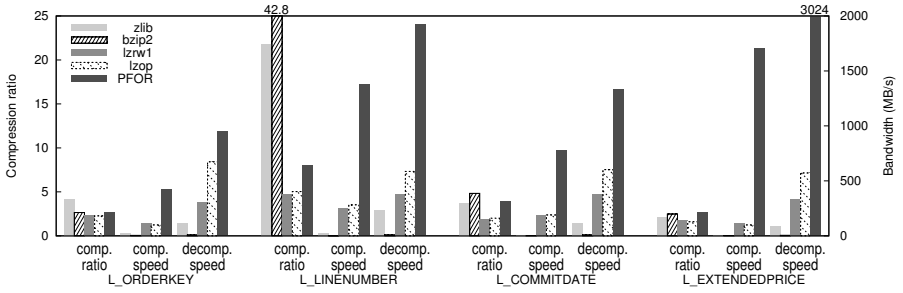


Figure 5.2: Comparison of various compression algorithms on a subset of TPC-H columns.

tions (*Delta Compression*). Such compression is the prime reason why inverted lists are now commonly considered superior to signature files as an IR access structure [WMB99]. Early inverted list compression work focused on exploiting the specific characteristics of gap distributions to achieve optimal compression ratio (e.g. using Huffman or Golomb coding tuned to the frequency of each particular term with a local Bernoulli model [Huf52]). More recently, attention has been paid to schemes that trade compression ratio for higher decompression speed [Tro03]. In Section 5.5, we show that our PFOR-DELTA compression scheme compares quite favorably with a speed-optimized compression scheme from the IR community, the word-aligned "carryover-12" [AM05b].

5.3 Super-scalar Compression

In this section we describe how insight in extracting high IPC (Instructions Per Cycle) efficiency from super-scalar CPUs led us to the design of PFOR, PFOR-DELTA and PDICT. Figure 5.2 shows that state-of-the-art "fast" algorithms such as LZRW1 or LZOP usually obtain 200-500MB/s decompression throughput on our evaluation platform (a 2.0GHz Opteron processor). However, we aim for 2-6GB/s.

Let us first motivate the need for such speed with the following simple model (all bandwidths in GB/s):

B = I/O bandwidth
r = compression ratio
Q = query bandwidth
C = decompression bandwidth
R = result tuple bandwidth

Our goal with compression is to make queries that are I/O bound (i.e. $Q > B$) faster:

$$R = \begin{cases} Br & : \quad \frac{Br}{C} + \frac{Br}{Q} \leq 1 \qquad \text{(I/O bound)} \\ \frac{QC}{Q+C} & : \quad \frac{Br}{C} + \frac{Br}{Q} \geq 1 \qquad \text{(CPU bound)} \end{cases} \tag{5.1}$$

Many datasets in e.g. data warehouses and information retrieval systems can be compressed considerably [GS91, GRS98]. Section 5.4 shows that even the synthetic TPC-H dataset, with its uniform distributions, allows for a good compression ratio. With these ratios, we often have $B < Q < Br$, such that

the query becomes CPU bound using compression. Also, modern RAID systems deliver $B > 0.3$GB/s, so with $r = 4$ one needs $C = 1.2$GB/s just to keep up with that. As we desire to spend only a minority of CPU time on decompression, we need $C = 2.4$GB/s to keep overhead to 50% of CPU time, and $C = 6$GB/s to get it down to 20%. These rules of thumb motivate our design goal of $C = 2{-}6$GB/s.

We must point out that achieving such high decompression bandwidth is hard. If we assume the decoded values to be 64-bit integers, e.g. $C = 3$GB/s means that 400M integers must be decoded per second, such that we can spend at most *five cycles per tuple* on our 2.0GHz machine! This motivates our interest in getting high IPC out of modern CPUs.

5.3.1 PFOR, PFOR-DELTA and PDICT

All our compression methods classify input values as either *coded* or *exception* values. Coded values are represented as small integers of arbitrary bit-width b, with $1 \leq b \leq 24$. The bit-width used for code values is kept constant within a disk block. Exception values are stored in uncompressed form, thus they should be infrequent in order to achieve a good compression ratio.

Our compression schemes are defined as follows:

PFOR Patched Frame-of-Reference: the small integers are positive offsets from a base value. One (possibly negative) base value is used per disk block. Unlike standard FOR, the base value is not necessarily the minimum value in the block, as values below the base can be stored as exceptions.

PFOR-DELTA PFOR on deltas: it encodes the *differences* between subsequent values in the column. Decompression consists of PFOR-decompression, and then computing the running sum on the result.

PDICT Patched Dictionary Compression. Integer codes refer to a position in an array of values (the dictionary). Not all values need to be in the dictionary; there can be exceptions. A disk block can contain a new dictionary but can also re-use the dictionary of a previous block.

The microbenchmarks presented throughout this section all compress 64-bit data items into 8 bits codes, but we implemented and tested our algorithms for all (applicable) datatypes and bit-widths b. In general, we found that, (de)compression bandwidth varies proportionally with the compression ratio.

Datasets encountered in practice are often skewed, both in terms of value distribution and frequency distribution. However, the existing FOR and dictionary compression cannot cope well with this. FOR compression needs $\lceil log_2(max - min + 1) \rceil$ bits, and is thus vulnerable to outliers if the data (i.e. value) distribution is skewed. In contrast, our new PFOR stores outliers as exceptions, such that the $[max_{coded}, min_{coded}]$ range is strongly reduced. Similarly, dictionary compression always needs $\lceil log_2(|D|) \rceil$ bits, even if the frequency distribution of the domain D is highly skewed. In PDICT, however, infrequent values become exceptions, such that the size $|D_{coded}|$ of the frequent domain is strongly reduced on skewed frequency distributions.

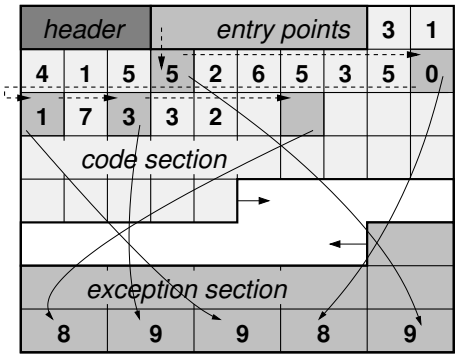


Figure 5.3: Compressed Segment Layout (encoding the digits of π: 3141**5**926535**8**97**9**32 using 3-bit PFOR compression, where digits greater than 7 – in bold – are encoded as exceptions).

5.3.2 Disk Storage

The minimum physical granularity for data storage on disk in ColumnBM is the *chunk*, which is a multiple of the filesystem page size, chosen such that sequential throughput on single-chunk requests approaches the disk bandwidth (depending on the hardware, values tend to range between 1MB and 8MB). Chunks contain one or more *segments*. In case of column-wise storage, a segment is identical to a chunk. In case of PAX [ADHS01], a chunk contains a segment for each column, and all segments in the chunk contain the same number of values, which implies that these segments may have different byte-sizes (that sum to a number close to the chunk size).

Uncompressed fixed-width data types are stored in a segment as a simple array of values.[1] Figure 5.3 shows the structure of a *compressed segment* that divides the segment in four *sections*:

- a fixed-size *header*, that contains compression-method specific info as well as the sizes and positions of the other sections.
- the *entry point section* that allows for fine-grained tuple access. For every 128 values, it contains an offset to the next exception in the code section, and a corresponding offset in the exception section.
- the *code section* is a forward-growing array with one small integer code for each encoded value. This section takes the majority of the space in the block.
- the *exception section*, growing backwards, stores non-compressed values that could not be encoded into a small integer code.

5.3.3 Decompression

A pre-processing step in decompression is *bit-unpacking*: the transformation of b bits-wide code patterns in the disk block into an array of machine-addressable

[1] Variable-width data types such as strings are stored in two segments: one byte-array that contains all values concatenated and a segment with integer offsets to their start positions.

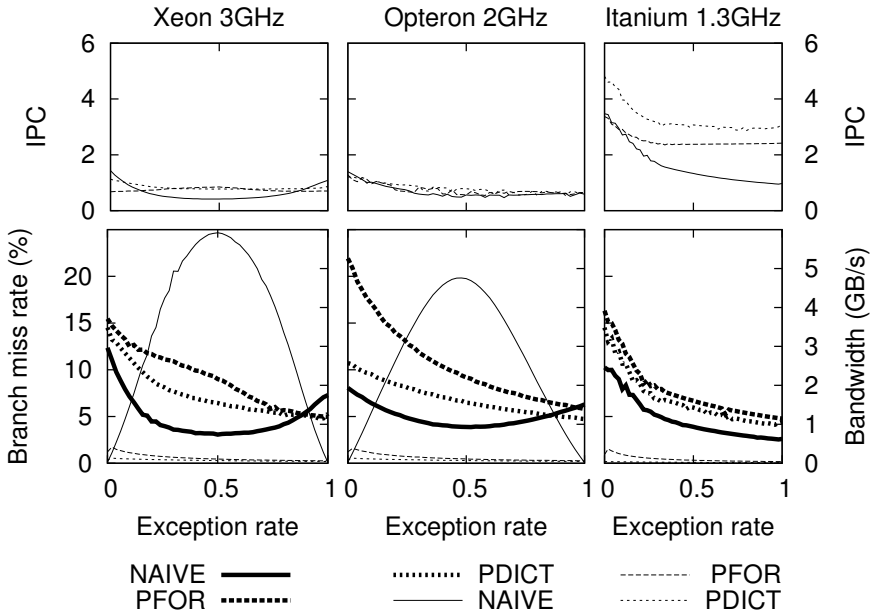


Figure 5.4: Decompression bandwidth (thick lines) and branch miss rate (thin lines) as a function of the exception rate.

integers (resp. bit-packing is post-processing for compression). It is done with highly optimized routines that are *loop-unrolled* to handle 32 values each iteration. We found this (un)packing to take up only a moderate fraction of our (de)compression cost, so we omit these details in our code.

The naive way to implement any decompression scheme that distinguishes between *coded* and *exception* values, is to use a special code (`MAXCODE`) for exceptions, and continuously test for it while decompressing:

```
/* NAIVE approach to decompression */
for(i=j=0; i<n; i++) {
  if (code[i] < MAXCODE) {
    output[i] = DECODE(code[i]);
  } else {
    output[i] = exception[--j]);
  }
}
```

The above decompression kernel is applicable to both PFOR and PDICT, though the way they encode/decode values differs. In our pseudo code, we abstract from these differences using the following macros: *(i)* `int ENCODE(ANY)`, that transforms an input value into a small integer, and *(ii)* `ANY DECODE(int)`, that produces the encoded input value given a small integer code.

The problem with the NAIVE approach is that it violates our guideline to avoid `if-then-else` in the inner loop. This hinders loop pipelining by the compiler, and also causes branch mispredictions when the else-branch is taken (assuming exceptions are the less likely event). The lower-left part of Figure 5.4 demonstrates most clearly on Pentium4 how NAIVE decompression throughput

rapidly deteriorates as the exception rate gets nearer to 50%. This is caused by branch mispredictions[2] on the `if-then-else` test for an exception, that becomes impossible to predict. In the graph on top, we see that the IPC (Instructions Per Cycle) takes a nosedive to 0.5 at that point, showing that branch mispredictions are severely penalized by the 31 stage pipeline of Pentium4.

To avoid this problem, we propose the following alternative "patch" approach:

```
int Decompress<ANY>( int n, int b,
  ANY   *__restrict__ output,
  void  *__restrict__ input,
  ANY   *__restrict__ exception,
  int   *next_exception )
{
  int next, code[n], cur = *next_exception;

  UNPACK[b](code, input, n); /* bit-unpack the values */

  /* LOOP1: decode regardless */
  for(int i=0; i<n; i++) {
    output[i] = DECODE(code[i]);
  }
  /* LOOP2: patch it up */
  for(int i=1; cur < n; i++, cur = next) {
    next = cur + code[cur] + 1;
    output[cur] = exception[-i];
  }
  *next_exception = cur - n;
  return i;
}
```

Different from the NAIVE method, decompression is now split in two tight loops without any `if-then-else` statements, that all can be loop-pipelined by a compiler.

Figure 5.3, depicting the integer sequence of π stored using 3-bit PFOR with $min_{coded} = 0$, shows that all exception values (i.e. digits ≥ 8) use their code value to store an offset to the next exception, forming a linked list.

The first loop simply decodes all values, which will generate wrong values for the exceptions. The second loop then *patches up* the incorrect values by walking the linked exception list and copying the exception values into the output array. The idea of patching rather than escaping exception values is central to our new algorithms, hence the "P" in their name derives from it.

Following the linked list during patching violates our guideline that one iteration should be independent of the previous one. Iterating the list poses a data hazard to the CPU, however, and not a control hazard, such that it is not very expensive. Moreover, the second loop processes only a small percentage of values, and the data it updates is in the CPU cache. That makes its overhead easily amortized by the performance improvement of the first loop.

The results in Figure 5.4 show that the performance of our patching algorithms decreases monotonically with increasing exception rates. Contrary to the NAIVE approach, decompression bandwidth degrades roughly proportionally with the compression ratio, or the size of the compressed data, as one

[2]We collected IPC, cache misses, and branch misprediction statistics using *CPU event counters* on all test platforms.

would expect. The relatively flat IPC lines suggest that the overhead of the data dependency in `LOOP2` is negligible with respect to the increase in memory traffic.

This does not hold for the NAIVE kernel, for which on Pentium4 and Opteron we observe a clear increase in decompression bandwidth towards an exception rate of one. This suggests that its performance is not determined by the size of the compressed data, but by branch mispredictions in the CPU, as both decompression bandwidth and IPC follow the inverse of the bell-shaped branch misprediction curve.

On Itanium2, the branch mispredictions are avoided thanks to branch predication explained in Section 2.4.8. As a result, the performance of the NAIVE kernel closely tracks that of PFOR and PDICT, as presented in the rightmost graph in Figure 5.4. Overall, the patching schemes are clearly to be preferred over the NAIVE approach, as they are faster on all tested architectures.

5.3.4 Compression

Previous database compression work mainly focuses on decompression performance, and views compression as a one-time investment that is amortized by repeated use of the compressed data. This is caused by the low throughput of compression, often an order of magnitude slower than decompression (see Figure 5.2), such that compression bandwidth is clearly lower than I/O write bandwidth. In contrast, our super-scalar compression *can* be used to accelerate I/O bound data materialization tasks. In OLAP and data mining environments, such materialization happens quite frequently for sorting, ad-hoc joins that require partitioning, or (view) materialization of intermediate results that are re-used by a subsequent query batch.

Efficient compression is also important for re-compression of data chunks after updates, as described later in Section 6.6. Note that I/O write bandwidth tends to be considerably lower than read bandwidth. Therefore, the design goal of compression throughput can be lower than for decompression, e.g. 1-2GB/s. The bottom graphs in Figure 5.5 show that PFOR compression meets this target on all our test platforms.

To achieve such high throughput, we again use the principle of avoiding `if-then-else` in the inner loop. The first loop uses a temporary array `miss` to make a list of exception positions. The second loop constructs the linked patch list and copies the exception values.

```
int Compress<ANY>( int n, int b,
  ANY  *__restrict__ input,
  void *__restrict__ code,
  ANY  *__restrict__ exception,
  int  *lastpatch
) {
  int miss[N], data[N], prev = *lastpatch;

  /* LOOP1: find exceptions */
  for(int i=0,j=0; i<n; i++) {
    int val = ENCODE(input[i]);
    data[i] = val;
    miss[j] = i;
    j += (val > MAXCODE);
  }
```

```
  /* LOOP2: create patchlist */
  for(int i=0; i<j; i++) {
    int cur = miss[i];
    exception[-i] = input[cur];
    data[prev] = (cur - prev) - 1;
    prev = cur;
  }
  PACK[b](code, data, n); /* bit-pack the values */
  *lastpatch = prev;
  return j; /* #exceptions */
}
```

Appending a position to the `miss` list without `if-then-else` uses a technique called *predication* (Section 2.4.8): the current position is always copied to the end of the list, and the list pointer is incremented with a boolean.

Predication transforms a control dependency into a data dependency, which is more efficient. Still, the presence of a data dependency on the variable `j` in the first, performance-critical loop, violates our guideline that iterations should be independent. Data dependencies cause delay slots in the CPU pipeline. The left-upper graph of Figure 5.5 shows that Pentium4 has an IPC of < 1. We can try to improve IPC by offering it more independent work using a technique called *double-cursor*. Here, the main loop, i.e. `LOOP1`, responsible for finding exceptions, runs two cursors through the to-be-encoded values, one from the start, and one from halfway. These cursors build two independent miss lists, containing the exception values that should be encoded in a post-processing step outside the main loop (using a slightly modified `LOOP2`, the code of which we omit).

```
/* LOOP1a: find exceptions */
int m = n/2;
for(int i=0, j_0=0, j_m=0; i<m; i++) {
  int val_0 = ENCODE(input[i+0]);
  int val_m = ENCODE(input[i+m]);
  code[i+0] = val_0;
  code[i+m] = val_m;
  miss_0[j_0] = i+0;
  miss_m[j_m] = i+m;
  j_0 += (val_0 > MAXCODE);
  j_m += (val_m > MAXCODE);
}
```

Double-cursor is not the same as loop-unrolling, and cannot be introduced automatically by the compiler.

Figure 5.5 shows that double-cursor significantly improves the IPC and throughput of PFOR on Pentium4, while it behaves the same as single-cursor PFOR on Opteron. On Itanium, where single-cursor already achieved a very high IPC (4), performance degrades somewhat. As the gains on Pentium4, which is also the more prevalent, outweigh the loses on Itanium, double-cursor can be considered the overall winner.

5.3.5 Fine-grained Access

While we anticipate that most performance-intensive queries will decompress *all* values in a compressed segment sequentially, some queries may perform random value accesses. A random lookup in the buffer manager will likely cause a CPU

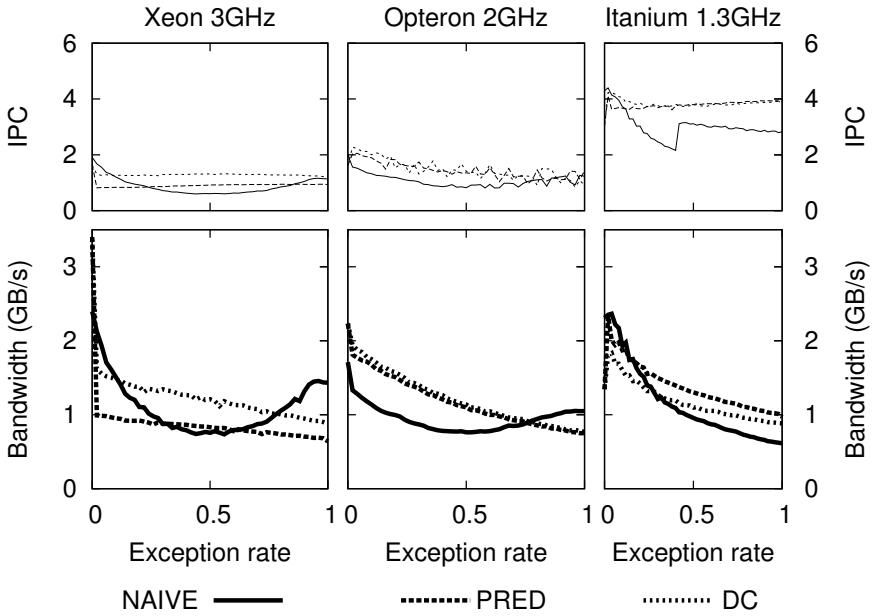


Figure 5.5: PFOR compression bandwidth as a function of exception rate for `if-then-else` (NAIVE), predication (PRED) and double-cursor (DC).

cache miss, so if decompression overhead stays in the same ballpark as DRAM access (i.e. 150-400 CPU cycles per cache miss), we deem it efficient enough.

It is easy to randomly access the `code` section at any position x, but we should also know whether position x is an exception and if so, where in the exception section the real value is stored. For this purpose, the *entry point section* keeps a pointer to the next exception, as well as its position in the exception section, for each position that is an exact multiple of 128. Each entry point, stored once every 128 values, is a combination of a 7-bits patch `start_list` and a 25-bits `start_exception`, hence the storage overhead of fine-grained access is $32/128 = 0.25$ bits per value. Note that 25-bits exception codes limit our segments to a maximum of 32MB, which is more than sufficient for now to obtain high sequential bandwidth on any RAID system. We can obtain the value at position x in the block, as follows:

```
ANY finegrained_decompress(int x,
  int*__restrict__ code,
  ANY*__restrict__ exception,
  entry_t*__restrict__ entry,
) {
  int i = entry[x>>7].start_list + x & ~127;
  int j = entry[x>>7].start_exception;
  while(i < x) {
    i += code[i]; j--;
  }
  return (i == x) ? exception[j] : DECODE(code[x]);
}
```

This tight pipelinable loop that walks the linked list takes 8,9 and 11 cycles

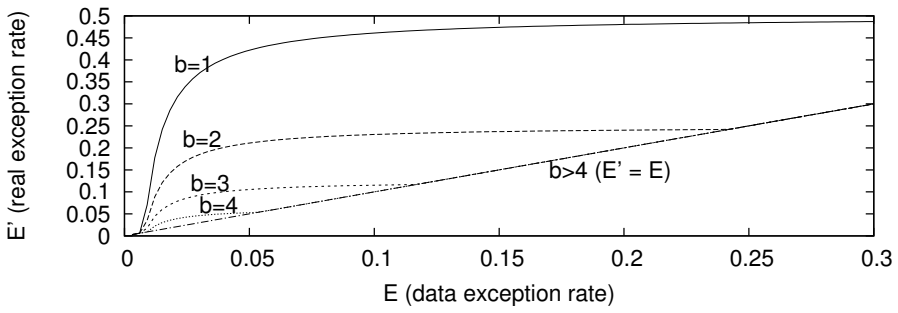


Figure 5.6: How compulsory exceptions increase the real exception rate E' for $b \leq 4$.

per iteration on respectively the Opteron, Itanium2 and Pentium4 CPUs. Even in the worst realistic case of 30% exceptions, it thus takes on average only a limited ($< 128 * 0.3/2 = 21$) number of iterations on average, such that random access decoding takes around 200 CPU work cycles per value.

In case of PFOR-DELTA, we must also store the current running total for each entry point. Sticking with 64-bit integers, this induces an additional storage overhead of 0.75 bit per value. Also, fine-grained PFOR-DELTA access requires decompressing a vector of 128 values (which usually causes one cache miss in both the code and exception sections, bringing memory access cost to 300-800 cycles). Since our decompression algorithms typically spend between 3-6 cycles per value, decompressing 128 values is in the same order of cost.

5.3.6 Compulsory Exceptions

A complication of patching is that the compressed integer codes only have a range from [0,2^b-1]; hence the maximum distance between elements in the linked list of exceptions is 2^b. If gaps exceeding this distance occur, so-called *compulsory exceptions* must be introduced. A compulsory exception is a value that can be compressed but is represented as an exception anyway, just in order to use its code value to keep the exception list connected.

We do not always have to insert compulsory exceptions if the gap is larger than 2^b though. Each *entry point* starts a new exception list, and these lists need not be connected to each other. Thus, gaps between exceptions at the start and end of each 128-value sequence never need compulsory exceptions. This effectively reduces the area in the code section that must be "covered" by a linked exception list per 128 values by $1/E$, where E is the exception rate caused by the data distribution. From this, we can compute E', which is the effective exception rate after taking into account compulsory exceptions as $E' = MAX(E, \frac{128E-1}{128E}2^{-b})$. Figure 5.6 shows that with bit-width $b = 1$ for miss rates $E > 0.01$, the effective exception rate E' quickly increases to a rather useless 0.47. With $b = 2$, it goes to an already more usable $E' = 0.22$, while for all bit-widths $b > 4$, the effect of compulsory exceptions is negligible.

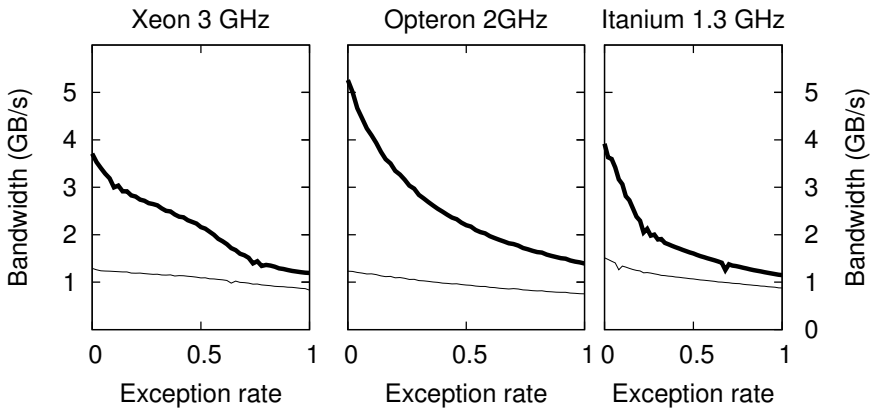


Figure 5.7: RAM-RAM (thin) versus RAM- Cache PFOR decompression (thick).

5.3.7 RAM-RAM vs. RAM-Cache Decompression

Figure 5.7 presents the results of a micro-benchmark conducted to evaluate our choice for fine-grained, into-cache decompression, as opposed to decompression on the granularity of disk pages. Into-cache decompression is achieved by decompressing a page on a per-vector basis, always storing the result in the same cache-resident result vector, overwriting any previous results. In the page-wise approach, the full, uncompressed page is materialized in RAM.

Results show that RAM-Cache decompression is much more efficient than RAM-RAM decompression. Performance of the former approach degrades with the exception rate, and thus the size of the compressed data. The flat shape of the latter approach suggests that performance is constrained by the need to materialize the uncompressed result, which is always constant in size.

Another benefit of the RAM-Cache approach is that the cache-resident result vector can be fed directly into an operator pipeline. In the RAM-RAM approach, the uncompressed page needs to be read back into the CPU, presenting an additional overhead which is not even incorporated in the RAM-RAM results from Figure 5.7.

5.3.8 Choosing Compression Schemes

The table materialization operator in Vectorwise should automatically decide which compression method to use for each disk chunk, and with what parameters. The idea is to first gather a sample (e.g. s=64K values) and look for the best settings for all applicable schemes. For numeric data types (e.g. integers, decimals) all three schemes apply. Otherwise, only PDICT is usable.

When a column is being compressed, the compression ratio can be easily monitored at the granularity of a disk chunk. When it strongly deteriorates, we could re-run the compression mode analysis to adapt the parameters for the next chunk or even choose another compression scheme. The complexity of choosing a compression mode is $O(s \log s)$ to the size of the sample s, because it

must be sorted as a preprocessing step. We now discuss for each method, how the optimal parameters are found using the sorted sample.

In PFOR, we can determine in one pass through the sorted sample where the longest stretch of values starts, such that the difference between first and last is representable in `b` bits.

```
PFOR_ANALYZE_BITS(int n, ANY *V, int b) {
  int len=0, min=0, range=1<<b;

  for(int lo=0, hi=0; hi<n; hi++)
    if (V[hi]-V[lo] >= range) {
      if (hi-lo > len) {
        min = lo; len = hi-lo;
      }
      while(V[hi]-V[lo] >= range) lo++;
    }
  return (min,len+1);
}
```

We simply invoke this function for all relevant bit-widths b and choose the setting that yields best compression, i.e. $1 \leq b < 8 * \texttt{sizeof(V)}$ where $b + E_{PFOR(b)} * 8 * \texttt{sizeof(V)}$ is minimal. In this equation the exception rate $E_{PFOR(b)} = \frac{s-len_b}{s}$, where len_b is returned by the above function, when invoked on the sample with parameter b.

The parameters for PFOR-DELTA are derived by running this same algorithm on the sorted differences of the sample.

For PDICT, we use once again the sorted sample, to create a (smaller) frequency histogram h, which we re-sort descending on frequency. PDICT will encode the first (i.e. largest) 2^b buckets of this histogram such that the exception rate $E_{PDICT(b)} = 1 - \sum_{i=1}^{2^b} \frac{h[i]}{s}$. Again, by trying all relevant settings of b, we can quickly determine the b that yields the highest compression rate. The first 2^b values from the histogram are subsequently used to create a super-scalar *memorized probing* hash function that is used during PDICT compression to compute the integer codes for values that must be compressed.

As illustrated in Figure 5.8, memorized probing aims to eliminate the iterations required to resolve collisions in traditional linear probing. During compression, it maintains two hash tables, a *value* dictionary of size M, which is indexed by the final codes and therefore densely packed to minimize their size, and a larger (e.g. $4M$) *offset* hash table, which is indexed by a distinct hash function and "memorizes" the small offsets (e.g. 1 byte) used to resolve collisions on the value hash. Using this approach, an array of values `v` can be encoded in a simple loop:

```
for(i=0; i<n; i++) {
   uint hash_idx = VALUE_HASH(v[i]);
   uint offset_idx = OFFSET_HASH(v[i]) & offset_mask;
   uint code = (hash_idx + offset_table[offset_idx]) & hash_mask;
   codes[i] = code;
   exceptions[j] = i;
   j += value_table[code] != v[i];
}
```

Memorized probing allows PDICT compression to achieve a compression bandwidth of $> 1GB/s$ on all our three test platforms.

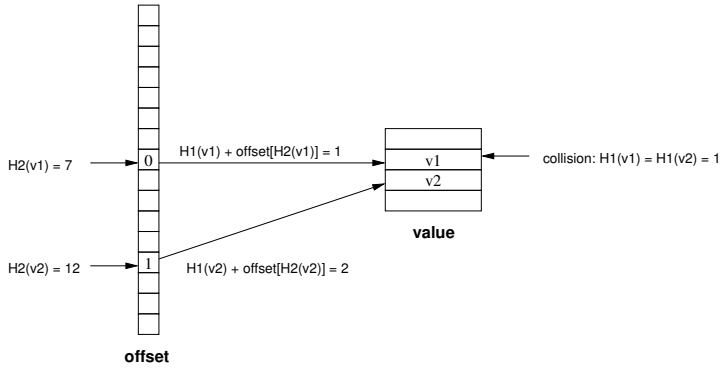


Figure 5.8: Memorized probing for M = 4. The offset hash, in combination with hash function H2, is used to memorize the offsets that are used to resolve collisions in the value hash, which uses hash function H1.

5.4 TPC-H Experiments

Table 5.1 shows the performance of our compression algorithms in Vectorwise running the TPC-H benchmark [Tra02] with scale factor 100 on two different hardware platforms. Low-end servers are represented by an Opteron 2GHz machine with a 4-disk RAID system delivering around 80 GB/s. The example of a middle-end system is a Pentium4 machine with 12-disk RAID delivering around 350GB/s. Both machines are dual CPU systems with 4GB memory, but here Vectorwise still only uses one CPU.

We used the same data clustering and index structures as in the previous in-memory Vectorwise TPC-H SF-100 experiments [BZN05]. Only a subset of TPC-H queries is presented, since, at the time of publication, the Vectorwise execution layer was missing some of the features necessary to run the remaining ones in a disk-based scenario.

While ColumnBM by default uses the DSM storage model [CK85], we also present the results for PAX storage [ADHS01]. I/O-wise they are comparable to an NSM system running DB2, for which the last column lists the official TPC-H scores. This system uses eight Pentium4 Xeon CPUs (2.8GHz), 16GB RAM and 142 SCSI disks. Thus, while the CPU used is roughly equivalent to our middle-end server, it has 4-12x more hardware resources across the board. We urge not to draw any further conclusions from these numbers other than that the Vectorwise results are in the high-performance ballpark.

The TPC-H data was compressed using PFOR, PFOR-DELTA and PDICT (enum) compression schemes. The second and third columns of Table 5.1 show the compression ratios achieved per query. Note, that since the "comment" attribute could not be compressed with our algorithms, the PAX queries achieve significantly lower compression ratios. Columns 4 and 10 show that in most cases we reach our decompression speed target of > 2 GB/s.

On the Opteron system, the speedup for most of the DSM queries is in line with the compression ratio. As the left part of Figure 5.9 shows, this is related to the fact that the low-end disk system makes the queries I/O-bound even with compression. The middle part of Figure 5.9 shows, that on the Pentium4 system with a faster RAID the situation is different with much higher CPU

TPC-H query	compression ratio		Pentium4 Xeon 3GHz 12-disk RAID 4GB RAM					
	DSM	PAX	dec.speed	DSM			PAX	
			MB/sec	unc.	$M \Rightarrow C$	$M \Rightarrow M$	unc.	$M \Rightarrow C$
1	2	3	4	5	6	7	8	9
01	4.33	2.30	4502	65.9	50.9	63.8	265.0	103.0
03	3.04	1.66	2306	8.9	6.0	7.1	45.5	27.0
04	8.15	1.82	3709	4.8	1.8	2.3	30.2	16.5
05	3.81	2.24	2421	17.2	16.2	16.7	81.2	36.7
06	4.39	2.25	2200	10.8	4.6	6.1	51.0	22.5
07	1.71	2.01	1457	34.4	40.8	48.3	158.0	76.5
11	2.14	1.08	4084	18.8	18.5	19.4	38.8	35.6
14	1.91	1.94	3688	5.8	4.9	5.4	22.1	11.5
15	2.70	2.13	2584	30.3	31.2	31.3	49.6	40.0
18	3.56	2.75	4315	38.9	13.6	21.3	419.9	151.5
21	4.11	2.12	2600	43.2	24.2	32.1	338.0	157.6

TPC-H query	compression ratio		Opteron 2GHz 4-disk RAID 4GB RAM					8 x P4 Xeon 2.8GHz 142 disks 16GB RAM
	DSM	PAX	dec.speed	DSM		PAX		IBM DB2
			MB/sec	unc.	$M \Rightarrow C$	unc.	$M \Rightarrow C$	UDB 8.1
1	2	3	10	11	12	13	14	15
01	4.33	2.30	3736	307.2	69.6	1098.9	480.3	111.9
03	3.04	1.66	2546	35.0	11.3	183.5	113.6	15.1
04	8.15	1.82	3018	18.2	2.4	115.5	65.9	12.5
05	3.81	2.24	2119	54.3	15.3	300.1	155.9	84.0
06	4.39	2.25	2031	48.2	10.7	232.7	104.3	17.1
07	1.71	2.01	1251	119.8	72.0	614.2	349.4	86.5
11	2.14	1.08	3225	27.0	14.6	180.9	162.2	19.5
14	1.91	1.94	2888	23.7	12.2	90.6	46.9	10.9
15	2.70	2.13	2464	44.9	22.4	209.8	97.1	21.6
18	3.56	2.75	3833	181.9	50.6	1379.7	704.9	318.2
21	4.11	2.12	2520	197.6	46.6	1423.5	759.2	374.9

Legend: *unc.* – uncompressed data
$M \Rightarrow C$ – memory-to-cache decompression
$M \Rightarrow M$ – memory-to-memory decompression

Table 5.1: TPC-H SF-100 experiments on Vectorwise (except DB2 results, taken from `www.tpc.org`)

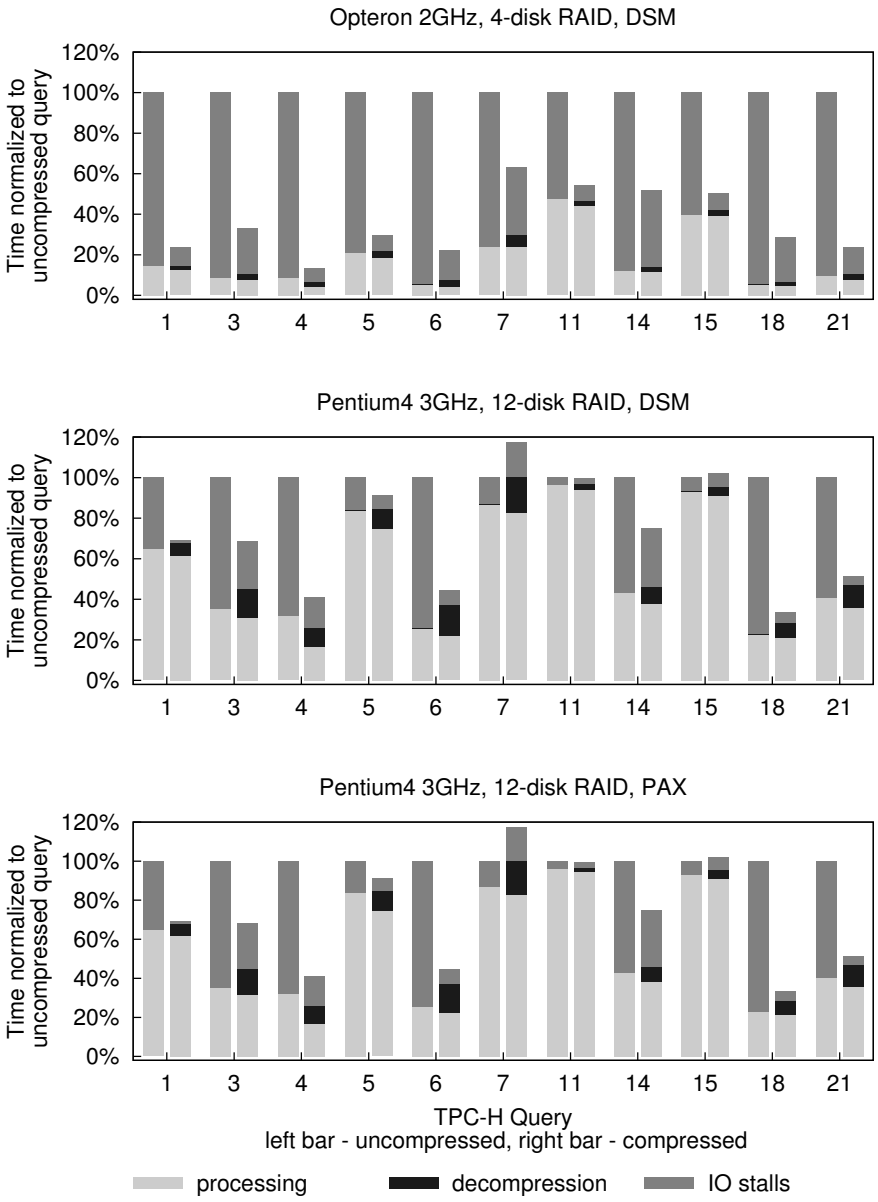


Figure 5.9: Profiling of TPC-H results from Table 5.1, showing processing, decompression and I/O time for uncompressed versus compressed runs.

	PFOR-DELTA			carryover-12			shuff		
	comp ratio	comp MB/s	dec MB/s	comp ratio	comp MB/s	dec MB/s	comp ratio	comp MB/s	dec MB/s
INEX	1.75	679	3053	2.12	49	524	2.45	3.5	82
TREC fbis	3.47	788	3911	4.26	98	740	5.11	190	164
TREC fr94	3.12	682	3196	3.49	84	689	4.65	149	154
TREC ft	3.13	761	3443	3.47	84	704	4.89	178	157
TREC latimes	2.99	742	3289	3.30	79	683	4.61	164	153

Table 5.2: PFOR-DELTA on Inverted Files.

usage in the uncompressed case. As a result, after increasing the perceived I/O bandwidth by decompression, all the queries become CPU-bound, such that the performance gain is less than the compression ratio. With the PAX storage model and its increased I/O requirements, the CPU processing impact is reduced again, resulting in better speedups than in the DSM case.

We also implemented the possibility to perform full-page decompression in ColumnBM. Column 7 of Table 5.1 shows that such decompression from memory into memory is significantly slower than the fine-grained decompression between RAM and the CPU Cache, presented in column 6. The main reason for this is the high-cost of in-memory materialization of decompressed data. Another interesting feature of our into-cache decompression can be observed in Figure 5.9, where the processing in the compressed case is slightly faster. This is caused by the fact that the main-memory access is performed by the decompression routines, and the query execution layer reads the data directly from the CPU cache.

5.5 Inverted File Compression

Compression of *inverted files* to improve I/O bandwidth and sometimes latency (due to reduced seek distances on the compressed file) is important for the performance of information retrieval systems [WMB99]. In this area, there is a trend to use lightweight-compression schemes rather than the classical storage-optimal schemes [Tro03, AM05b].

We evaluated the performance of PFOR-DELTA with respect to both compression ratio and speed on inverted file data derived from the INEX and TREC document collections, and compared it against *carryover-12* [AM05b], a compression scheme designed for fast decompression of inverted files. Furthermore, performance was compared to that of a semi-static Huffman coder, which is commonly used for inverted file compression. Table 5.2 summarizes the results on our 3GHz Pentium 4 machine, and shows that PFOR-DELTA improves decompression bandwidth of *carryover-12* 6.5 times, while only reducing the compression ratio by 15%.

To verify the need for such decompression speeds, we measured the raw query bandwidth of a typical retrieval query that looks up the top-N documents in which a given term from the TREC fbis dataset occurs most frequently (see Section A.3). Within our Vectorwise system, this query was able to process a list of d-gaps at 580MB/s, which implies that even on our 350MB/s RAID system it would remain I/O-bound. Using equation 5.1 to compute the decompression bandwidth C that achieves an equilibrium between CPU time spent

on query processing and decompression, yields $\frac{580 \times C}{580+C} = 350$, which leads to $C = 883$MB/s. Table 5.2 shows that decompression bandwidths from *shuff* and even *carryover-12* are below this point, hence only make the query slower, while PFOR-DELTA accelerates it from 350MB/s to 504MB/s.

5.6 Conclusions and Future Work

This chapter presented our work on using data compression to scale the high performance of the Vectorwise engine to disk-based datasets. We proposed a new set of *super-scalar* compression algorithms. The use of "patching" allows those to handle outliers gracefully, while still utilizing the pipelined features of modern CPUs. Additionally, we introduced the idea of decompressing between RAM and the CPU Cache, rather than the common idea to apply it between I/O and RAM. Our results show that this not only allows the buffer manager to store more (compressed) data, but is also faster to (de)compress. As a result, our algorithms provide decompression speeds in the range of $> 2GB/s$. This is an order of magnitude faster than conventional compression algorithms, making decompression almost transparent to query execution. By using these techniques in TPC-H, TREC and INEX datasets, we managed to significantly reduce or completely eliminate the I/O bottleneck.

Chapter 6

Positional Updates

6.1 Introduction

Chapter 5 has shown how columnar storage (DSM), in combination with light-weight compression, can be leveraged to improve I/O bandwidth utilization over a row-wise (NSM/PAX) approach to storage. This is particularly useful for data-intensive, read-mostly workloads, as found in, for example, data science and business intelligence. However, such read-optimized columnar storage is notoriously write-unfriendly. Not only does the in-place insertion or deletion of a single tuple within the stored data introduce I/O proportional to the number of attributes, but it can be further complicated by a read-optimized page layout. For example, pages in a column-store are often large, to optimize for sequential I/O, and densely packed with compressed data that often adheres to strict ordering or clustering constraints. Therefore, even inserting into a single page might call for decompression, re-compression after modification, and potentially for large amounts of densely packed, ordered tuples to be shifted.

In this chapter we address the question how read-optimized column-store systems can still efficiently provide update functionality, without compromising their read performance. To avoid operating directly on the physical data, we propose to use a differential approach to updates, which, by itself, is not new. The novelty comes from the fact that we organize updates by tuple *position*, i.e. the offset of a tuple within a table. To efficiently handle tuple positions, which are volatile in nature, this chapter introduces a novel data-structure, which we call *positional delta tree* (PDT). Furthermore, we show how this data-structure can be integrated into a fully transactional DBMS architecture.

6.1.1 Differential Updates

Some analytic columnar database systems, such as C-Store [Sto05], handle updates by splitting their architecture in a *read-store* (RS) that manages the bulk of all data and a *write-store* (WS) that manages the most recent updates. Consequently, all queries access both base table information from the read-store, as well as all corresponding differences from the write-store and *merge* these on-the-fly. Also, in order to keep the write-store small (it resides typically in RAM), changes in it are periodically propagated into the read-store [Sto05].

The topic of this chapter is what data structures and algorithms should be used for implementing the write-store of a column-oriented database system that aims to support generic update workloads (inserts, deletes, modifications; batch or trickle). The natural "value-based" way to implement the write-store is to keep track of which tuples were inserted, deleted or modified in a RAM-resident data structure that organizes these items in the *sort key* (SK) order of the underlying read-store table and contains these keys; for example in a RAM friendly B-tree [CGM01]. Such a data structure can easily be modified for update queries, and can also be merged efficiently with the read-store for read-queries by scanning the leaves. An important side effect of the value-based approach, however, is that all queries must scan all sort key columns in order to perform the merge (even if those queries themselves do not need the keys) which reduces an important advantage of column stores.

6.1.2 Positional Updates (PDT)

In this chapter, we propose a new data structure called the Positional Delta Tree (PDT). PDTs are similar to counted B-trees [Tat01] but contain differential updates. They are designed to make merging in of these updates fast by providing the tuple *positions* where differences have to be applied at update time. Thanks to that, read queries that merge in differences do not need to look at sort key values. Instead, the merge operator can simply count down to the next position where a difference occurs, and apply it blindly when the scan cursor arrives there.

The key advantages of the PDT over value-based merging are:

1. positional merging needs less I/O than value-based merging, because the sort keys do not need to be read.

2. positional merging is less CPU intensive than value-based merging, especially when the sort-key is a compound key (common in clustering approaches) and/or non-numerical attributes are part of the sort-key.

3. positional updates allow for an efficient implementation of modifies, encoding them as a delta in a two-dimensional grid, rather than deleting the original tuple and reinserting it with one or more attributes modified.

4. positional updates allow for an efficient implementation of a "block delete", i.e. a consecutive range of deleted tuples, by encoding them as a pair of $(offset, count)$.

5. positional updates require, in general, a smaller memory footprint than value-based approaches, which need to organize deltas by attribute values, rather than a single positional index.

While the concept of positional differences sounds simple, its realization is complex, since positions in an ordered table are highly volatile: after an insertion or deletion, all positions of subsequent tuples change. The core of the problem is managing a mapping between two monotonically increasing numbers associated with tuples in a table, called the *stable ID* (SID) and current *row ID* (RID). The SIDs conceptually correspond to the consecutively ascending tuple

positions found in the read-store, but are in the current (i.e. "up-to-date") tuple order not necessarily consecutive nor unique, just non-decreasing.

As PDTs capture updates, they also form an important building block in transaction management. We show that three layers of PDTs can be used to provide *lock-free* isolation. This lock-free property is a great advantage for an analytic DBMS designed not to compromise read-query performance for updates. We define the basic notions on which to base ACID properties of PDT based transactions, and provide two PDT transformation algorithms (Propagate and Serialize), pivotal for transaction management.

All in all, the main contribution of this chapter is the PDT data structure and its application in column stores for providing efficient transactional updates without compromising read performance.

6.1.3 Outline

We start with some basic terminology in Section 6.2. The PDT data structure itself is introduced in Section 6.3, where its working is illustrated by examples. Detailed algorithms for PDT update and MergeScan operators are provided in Section 6.4. In Section 6.5 we then investigate PDT-based transaction management, using a layered approach to snapshot isolation to implement efficient optimistic concurrency control. The propagation of PDT updates to disk, which we call "checkpointing", and write-ahead-logging, are discussed in Section 6.6. We evaluate the performance of PDT based update management in Section 6.7, both using micro benchmarks, as well as in a TPC-H 30GB comparison between read-only, value-based and PDT-based query processing under an update load. Finally, in Section 6.8 we describe related work.

6.2 Terminology

6.2.1 Ordered Tables

A column-oriented definition of the relational model is as follows: each column col_i is an ordered sequence of values, a TABLE$<\text{col}_1, \cdots, \text{col}_n>$ is a collection of related columns of equal length, and a *tuple* τ is a single row within TABLE. Tuples consist of values aligned in columns, i.e., the attribute values that make up a tuple are retrievable from a column using a single positional index value. This index we call the *row-id*, or RID, of a tuple.

Though the relational model is order-oblivious, column-stores are internally conscious of the physical tuple ordering, as this allows them to reconstruct tuples from columns cheaply, without expensive value-based joins [RDS03]. The physical storage structures used in a column-store are designed to allow fast lookup and join by position using either B-tree storage with the tuple position (RID) as key in a highly compressed format [Gra07] or dense block-wise storage with a separate sparse index with the start RID of each block.

Columnar database systems not only exploit the fact that all columns contain tuples in some physical order that is the same for all of them, but often also impose a particular physical order on the tuples, based on their value. That is, tuples can be ordered according to sequence of sort attributes S, typically determined by the DBA [Sto05]. A sequence of attributes that defines a sort

order, while also being a key of a table we call the *sort key*, or SK. The motivation for such ordered storage, is to restrict scans to a fraction of the disk blocks if the query contains range- or equality-predicates on any prefix of the sort key attributes. As such, explicitly ordered storage is the columnar equivalent of index-organized tables (clustered indices) also used in a row-oriented RDBMS.

6.2.2 Ordering vs. Clustering

Multi-column sort orders are closely related to table clustering, that organizes a large (fact) table in groups of tuples, each group representing a cell in a multi-dimensional cube, where those dimensional values are typically reachable over a foreign-key link. Such multi-dimensional table clustering is an important technique to accelerate queries in data warehouses with a star- or snowflake-schema [PBM+03, COO08]. In a column-store, multi-dimensional clustering translates into ordering the tuples in a table in such a way that tuples from the same cell are stored contiguously. This creates the same situation as in ordered columnar table storage, where physical tuple position is determined by the tuple values; and the machinery to handle table updates needs to respect this value-based tuple order.

6.2.3 Positional Updates

An update on an ordered table is one of *insert*, *delete* or *modify*, defined as:

TABLE.insert(τ, i) adds a full tuple τ to the table at RID i, thereby incrementing the RIDs of existing tuples at RIDs $i \cdots N$ by one.

TABLE.delete(i) deletes the full tuple at RID i from the table, thereby decrementing the RIDs of existing tuples at RIDs $i + 1 \cdots N$ by one.

TABLE.modify(i, j, v) changes attribute j of an existing tuple at RID i to value v.

Multiple subsequent updates can be grouped into a single, atomic *transaction*.

6.2.4 Differential Structures

We focus on the situations where positional updates happen truly scattered over the table. Note that in ordered table storage, even append-only warehouse update workloads drive column-stores into worst-case territory if an in-place update strategy would be used, creating an avalanche of random writes, one for each affected row *and* column (modifies and deletes behave similarly). For this reason, and as argued previously, column-stores must use some differential structure DIFF, that buffers updates that have not yet been propagated to a stable table image.[1] As a consequence, all queries must not just scan stable table data from disk, but also apply the updates in DIFF as tuples are produced, using a Merge operator.

[1] Just like row-stores, at each commit column-stores need to write information in a Write-Ahead-Log (WAL), but that causes only sequential I/O, and does not limit throughput.

6.2.5 Checkpointing

Differential updates need to be eventually propagated to the stable storage to reduce the memory consumption and merging cost. While many strategies for this process can be envisioned, the simplest one is to detect a moment when the size of DIFF starts to exceed some threshold and to create a new image of the table with all updates that happened before applied. When this is ready, query processing switches over to this new image with the applied updates pruned from DIFF.

6.2.6 Stacking

Previous work on differential structures, e.g. differential files [SL76] and the Log-Structured Merge Tree [OCGO96] suggests *stacking* differential structures on top of each other, typically with the smallest structure on top, increasing in size towards the bottom. One reason to create such a multi-layer structure is to represent multiple different database snapshots, while sharing the bulk of the data structures (the biggest, lowest layers of the stack). Another reason is to limit the size of the topmost structure, which is the one being modified by updates, thus providing a more localized update access pattern, e.g. allowing to store different layers of the stack on different levels of the memory hierarchy (CPU caches, RAM, Flash, Disk). Each structure $\text{DIFF}_{t_2}^{t_1}$ in the stack contains updates from a time range $[t_1, t_2\rangle$:

$$\text{TABLE}_{t_2} = \text{TABLE}_{t_1}.\text{Merge}(\text{DIFF}_{t_2}^{t_1}) \tag{6.1}$$

If one keeps not one, but a stack of DIFFs (most recent on top), we can see the current image of a relational table as the result of merging the DIFFs in the stack one-by-one bottom-up with the initial TABLE_0. Here, TABLE_0 represents the "stable table", i.e. the initial state of a table on disk when instantiated (empty), bulk-loaded, or checkpointed.

Definition 1. *Two differential structures are* **aligned** *if the table state they are based on is equal:*
$Aligned(DIFF_{t_b}^{t_a}, DIFF_{t_d}^{t_c}) \Leftrightarrow t_a = t_c$

Definition 2. *Two differential structures are* **consecutive** *if the time where the first difference ends equals the time the second difference starts:*
$Consecutive(DIFF_{t_b}^{t_a}, DIFF_{t_d}^{t_c}) \Leftrightarrow t_b = t_c$

Definition 3. *Two differential structures are* **overlapping** *if their time intervals overlap:*
$Overlapping(DIFF_{t_b}^{t_a}, DIFF_{t_d}^{t_c}) \Leftrightarrow t_a < t_d \leq t_b \vee t_c < t_b \leq t_d$

The time t rather than absolute time identifies the moment a transaction started, and could be represented by a monotonically increasing logical number, such as a Log Sequence Number (LSN). If a query that started at t works with just a single DIFF_t^0 structure, we shorten our notation to DIFF_t.

6.2.7 RID vs. SID

We define *stable-id* $\text{SID}(\tau)$, to be the position of a tuple τ within the TABLE_0. (i.e. the "stable" table on disk) starting the tuple numbering at 0, and define

the *row-id* $\text{RID}(\tau)_t$, to be the position of τ at time t; thus $\text{SID}(\tau) = \text{RID}(\tau)_0$. $\text{SID}(\tau)$ never changes throughout the lifetime of tuple τ (except for checkpoints). We actually define $\text{SID}(\tau)$ to also have a value for newly inserted tuples τ: they receive a SID such that it is larger than the SID of all preceding stable tuples (if any) and equal to the first following stable tuple (if any). Conversely, we also define a RID value for a stable tuple that was deleted ("ghost tuples") to be one more than the RID of the preceding non-ghost tuple (if any) and equal to the first following non-ghost tuple (if any).

If the tuple τ is clear from the context, we abbreviate $\text{SID}(\tau)$ to just SID (similar for RID), and if the time t is clear from the context (e.g. the start time of the current transaction) we can also write just RID instead of RID_t. In general, considering the stacking of PDTs and thus having the state of the table represented at multiple points in time, we define Δ (*"delta"*) as the RID difference between two time-points:

$$\Delta_{t_2}^{t_1}(\tau) \quad = \text{RID}(\tau)_{t_2} - \text{RID}(\tau)_{t_1} \tag{6.2}$$

which in the common case when we have one PDT on top of the stable table ($t_1 = 0$) is RID minus SID:[2]

$$\Delta_t(\tau) \quad = \text{RID}(\tau)_t - \text{SID}(\tau) \tag{6.3}$$

If we define the SK-based Table time-wise difference as:

$$\text{MINUS}_{t_2}^{t_1} \quad = \{\tau \in \text{TABLE}_{t_1} : \nexists \gamma \in \text{TABLE}_{t_2} : \tau.\text{SK} = \gamma.\text{SK}\} \tag{6.4}$$

then we can compute Δ as the number of inserts minus the number of deletes before τ:

$$\begin{aligned} \Delta_{t_2}^{t_1}(\tau) \quad = \quad & |\{\gamma \in \text{MINUS}_{t_1}^{t_2} : \text{RID}(\gamma)_{t_2} < \text{RID}(\tau)_{t_2}\}| - \\ & |\{\gamma \in \text{MINUS}_{t_2}^{t_1} : \text{SID}(\gamma) < \text{SID}(\tau)\}| \end{aligned} \tag{6.5}$$

Note that the expression before the subtraction represents the number of tuples *inserted* between t_1 and t_2 for which $\text{RID}(\gamma)_{t_2} < \text{RID}(\tau)_{t_2}$. This is due to the MINUS being "reversed", i.e. $\text{MINUS}_{t_1}^{t_2}$ rather than the $\text{MINUS}_{t_2}^{t_1}$, which we use to represent deleted tuples between t_1 and t_2.

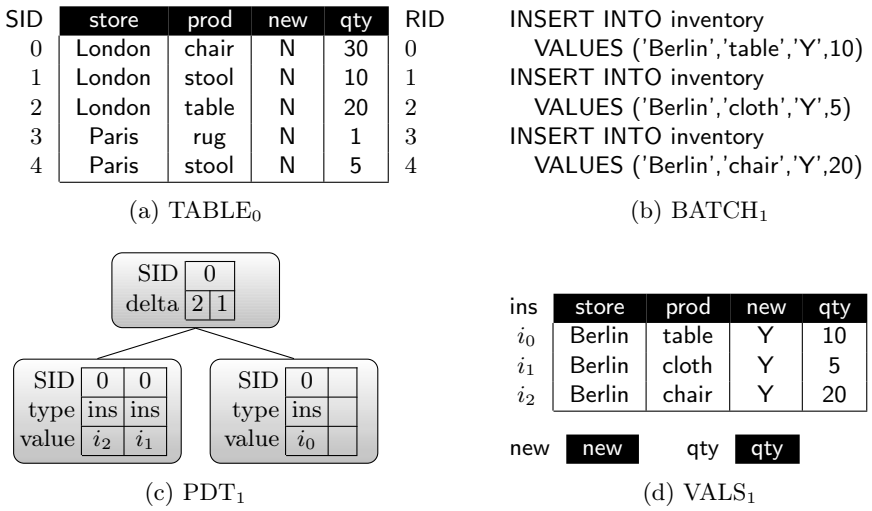
6.3 PDT by Example

We introduce the Positional Delta Tree (PDT) using a running example of a data warehouse table inventory, with sort key (store,prod), shown in Figure 6.1a. This TABLE_0 is already populated with tuples, using an initial bulk load, and persistently stored on disk. Initially its RIDs are identical to the SIDs. Note that RIDs and SIDs are conceptual sequence numbers; they are not stored anywhere.

6.3.1 Inserting Tuples

We now execute the insert statements from Figure 6.1b and observe the effects to both the PDT tree structure in Figure 6.1c and the *value space*, which holds

[2] When the context $\text{PDT}_{t_2}^{t_1}$ is clear, we sometimes slightly imprecisely refer to RID_{t_1} as SID and RID_{t_2} simply as RID, even in case of a stacked PDT ($t_1 > 0$).



(a) TABLE_0 (b) BATCH_1

(c) PDT_1 (d) VALS_1

Figure 6.1: Insert tuples into empty PDT.

the updated tuple data referenced by the PDT leaf nodes, in Figure 6.1d. Because the inventory table is kept sorted on (store,prod), the new Berlin tuples get inserted at the beginning. Rather than touching the stable table, the updates are recorded in a PDT shown in Figure 6.1c, which is a B^+-tree like structure that holds its data in the leaves.[3] The non-leaf nodes of the PDT (here only the root node) contain a SID as separator key, and a delta (explained below) that allows to compute the RID⇔SID mapping, because after the inserts these will no longer be identical. The separator SID in the inner nodes is the minimum SID of the right subtree. Figure 6.1c shows that the inserts all get the same SID 0, which is thus not unique in the PDT. Among them, the left-to-right leaf order in the PDT determines their order in the final results, which is displayed in Figure 6.2a.

6.3.2 Storing Tuple Data

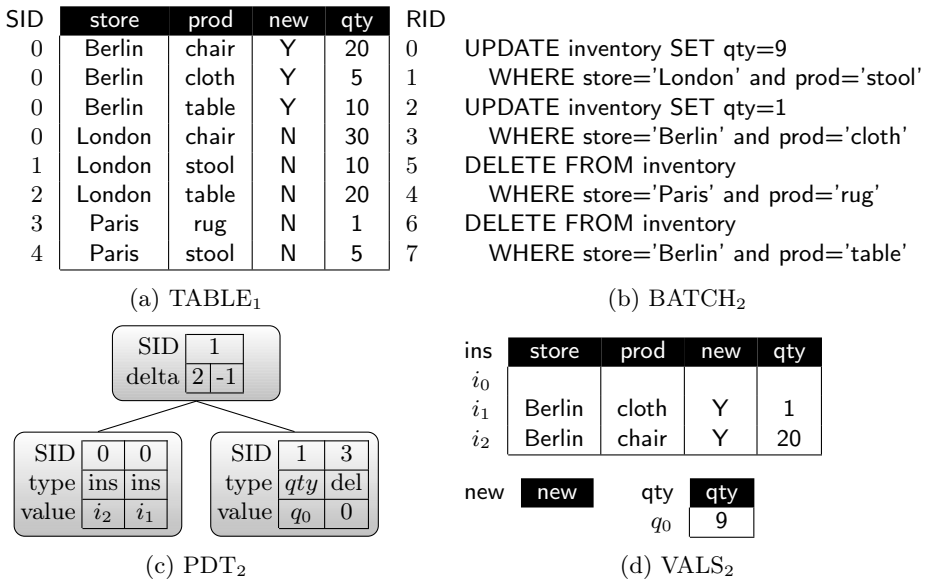
The leaf nodes of the PDT store the SID where each update applies, as well as the type of update, as well as a reference (or pointer) to the new tuple values. Because the type of information to store for insert and modify differs, these are stored in separate "value tables". In terms of our notation, we have:

$$\text{DIFF} = (\ \text{PDT}, \text{VALS}\ ) \tag{6.6}$$

$$\text{VALS} = (\ \text{ins}\langle \text{col}_1, \cdots, \text{col}_n\rangle, \ col_1\langle \text{col}_1\rangle, \cdots, col_n\langle \text{col}_n\rangle\ ) \tag{6.7}$$

Thus, each PDT has an associated "value space" that contains multiple value tables: one insert-table with new tuples and for each column a single-column modify-table with modified attribute values. The value space resulting from the insert statements in Figure 6.1d has all tables empty except the insert table.

[3]The tree fan-out is 2 for presentation purposes. Given its use as a cache/memory-resident data structure, node sizes should be a few cache lines long, e.g. a fan-out of 8 or 16.



(a) TABLE_1 (b) BATCH_2

(c) PDT_2 (d) VALS_2

Figure 6.2: Adding modifies and deletes to PDT.

6.3.3 Modifying Attribute Values

Moving to Figures 6.2a-6.2d we show the effects of modify (i.e. SQL UPDATE) and delete statements. The first update statement modifies the *qty* column of a stable tuple from TABLE_1. Such a modification gets encoded as a PDT update involving SID 1. The type field of this leaf entry encodes an identifier for the modified column, while the value field stores the value offset within that column. In the value space, each column has a separate, single-column table that holds modified values. In our PDT examples, we indicate modifications using the column name in italics (*qty*). Thus, the first value of the right leaf states that stable tuple SID=1 (i.e. (London,stool,N,10)) had its qty column set to 9, per the q_0 reference to the qty modification table in the value space of Figure 6.2d. Note that the key surrogates of the value space tables e.g. i_1, q_0 are not drawn as part of those tables, as simple numerical offsets can be used to save memory in the value space.

The second modify, changing the *qty* column of the (Berlin,cloth) tuple, involves a tuple that already resides in the PDT, as it was inserted during BATCH_1. In this case, we do not add a modify leaf entry to the PDT, but rather apply the change "in-place" to the memory resident data. We can see that the *qty* column of the i_1 tuple within VALS_2 has been changed to a value of 1 in Figure 6.2d.

Finally, note that the value space for modifies does not contain the sort-key columns (store,prod). This is the case because modification of a sort-key attribute of tuple τ is handled as a deletion of τ, followed by an insert of the modified tuple, as modification of sort-key attributes generally changes the sort order position of a tuple.

6.3.4 Deleting Tuples

Deletions produce a leaf node with the "del" type and the SID identifying the underlying tuple to be deleted. Instead of a reference to value space tuple data, deletes contain a *count* of *subsequent* tuples to be deleted. This is used to implement *block deletes* of consecutive tuples. In case of the default, deletion of a single tuple, this count is 0. The first DELETE statement, of the (Paris,rug) tuple, therefore gets encoded in the PDT leaf as a delete entry with SID 3 and count 0. In Figure 6.3a we still display this deleted tuple, but grayed out, as it is not visible anymore to queries.

Note that the second DELETE, for (Berlin,table), involves a tuple that is not stable (i.e. in TABLE_0) and can therefore be deleted "in-place" directly from the memory resident PDT structure (such tuples do not have a unique SID, so referencing them from the PDT leaves would be ambiguous). Therefore, the second delete removes all traces of the i_0 insert (Berlin,table,Y,10) from the PDT structure.

In a scenario where we delete a stable tuple which has modify entries for some of its attributes in the PDT, all such entries are removed and replaced by a single deletion entry (not shown in this example).

6.3.5 RID $\Leftrightarrow$ SID

Figures 6.3a-6.3d show the effect of three additional inserts, producing a final table state of Figure 6.3e. Of course, what is stored on disk is still TABLE_0 shown in Figure 6.1a. At this stage, the PDT has grown to a tree of three levels. To illustrate the way RIDs are mapped into SIDs, the PDT_3 in Figure 6.3c is annotated with the running Δ, as well as with the RID. Note that Δ, which is the number of inserts minus the number of deletes so far, as defined in Equation 6.5, takes into account the effect of all *preceding* updates, hence it is still zero at the leftmost insert. For the inner nodes, the annotated value is the separator value; the lowest RID of the right subtree. The PDT maintains the delta field for each child pointer in an inner node, which is the contribution to Δ of all updates in the respective subtree below it, i.e. the difference between the number of inserted and the number of deleted tuples within that subtree. Note that this number can be negative, as in Figure 6.2c.

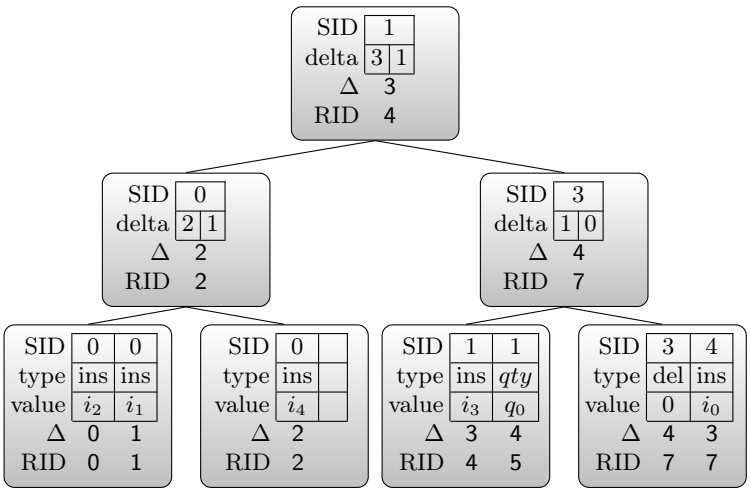
An important property of the PDT is that we can determine the Δ (and thereby RID) of the first entry in each leaf without performing a linear scan over all leaf entries that precede it (to count inserts and deletes). Rather, during a root-to-leaf traversal, within each internal PDT node, we sum the delta values belonging to each child pointer to the *left* of the pointer we eventually follow, thereby summing the cumulative deltas of each subtree that we keep to our left. This gives us the delta contribution of all leaves to the left of our final leaf. By counting inserts and deletes from left-to-right in the final leaf, we can compute Δ, and thus RID, for any leaf entry. Since lookup as well as updates to the PDT only involve the nodes on a root-to-leaf path, cost is logarithmic in PDT size.

SID	store	prod	new	qty	RID
0	Berlin	chair	Y	20	0
0	Berlin	cloth	Y	1	1
0	London	chair	N	30	2
1	London	stool	N	9	3
2	London	table	N	20	4
3	Paris	rug	N	1	5
4	Paris	stool	N	5	5

(a) TABLE_2

```
INSERT INTO inventory
   VALUES ('Paris','rack','Y',4)
INSERT INTO inventory
   VALUES ('London','rack','Y',4)
INSERT INTO inventory
   VALUES ('Berlin','rack','Y',4)
```

(b) BATCH_3



(c) PDT_3 with annotated Δ and RID=SID+Δ.

ins	store	prod	new	qty
i_0	Paris	rack	Y	4
i_1	Berlin	cloth	Y	1
i_2	Berlin	chair	Y	20
i_3	London	rack	Y	4
i_4	Berlin	rack	Y	4

new	new

qty	qty
q_0	9

(d) VALS_3

SID	store	prod	new	qty	RID
0	Berlin	chair	Y	20	0
0	Berlin	cloth	Y	1	1
0	Berlin	rack	Y	4	2
0	London	chair	N	30	3
1	London	rack	Y	4	4
1	London	stool	N	9	5
2	London	table	N	20	6
3	Paris	rug	N	1	7
4	Paris	rack	Y	4	7
4	Paris	stool	Y	4	8

(e) TABLE_3

Figure 6.3: Adding another batch of PDT inserts into TABLE_2.

.

store	prod	new	qty
Berlin	chair	Y	20
Berlin	cloth	Y	1
Berlin	rack	Y	4
London	rack	Y	4
London	stool	N	9
Paris	rack	Y	4

(a) Insertion table

store	prod
Paris	rug
London	stool

(b) Deletion table

Figure 6.4: Value-based differential updates.

6.3.6 Value-based Delta Trees (VDTs)

Other column-stores that use differential update processing, like MonetDB [Bon02], use a simpler "value-based" approach, that consists of an insert table that contains all columns and holds all inserted and modified tuples, and a deletion table, that only holds the sort key values of deleted or modified tuples. This is illustrated in Figure 6.4. Note that both tables are kept organized in sort key order, to facilitate merging these delta tables with the stable table. Therefore, it is natural to implement such tables as B^+-trees, hence we call this approach the *Value-based Delta Tree* (VDT). A VDT based DBMS should replace each table (range) scan of our example inventory table by:

```
SELECT * FROM ins
UNION
(SELECT * FROM inventory WHERE NOT EXISTS
   (SELECT * FROM del
    WHERE inventory.store = del.store AND inventory.prod = del.prod))
```

While the above seems intimidating, the fact that all three tables are kept organized in the order of the sort key, allows efficient evaluation using a physical relational algebra plan based on linear merge-union and -difference operators:

```
MergeUnion[store,prod](
   Scan(ins),
   MergeDiff[store,prod](
      Scan(inventory),
      Scan(del)
   )
)
```

6.3.7 Merging: PDT vs VDT

The main disadvantage of the VDT approach is that merging in of updates involves Merge-Union/-Diff comparisons on the sort keys (here (store,prod)), which is *(i)* computationally intensive, and *(ii)* forces the database system to read those sort keys off disk for every query, even in cases where the query does not involve these keys. The positional-only merging that PDTs offer, on the other hand, avoids reading key columns in these cases, an important advantage in a column-store. An added benefit is that PDTs allow for precise and efficient encoding of modify updates.

Similar benefits exist during execution of an update query itself. When deleting or modifying existing tuples, the PDT approach allows us to restrict

scans to attributes that are present in the update query plan, avoiding columns to be read for the sole purpose of retrieving attribute values to copied to the VDT.

Some systems (e.g. Vertica) employ optimizations improving the performance of value-based delta structures. For example, it is often possible to perform (parts of) a query on the stable and delta data separately, and only combine the results at the end. Also, deletions can be handled by using a Boolean column marking the "alive" status of a given tuple, stored in RAM using some updateable variant of the compressed bitmap index (a non-trivial data structure, see e.g. [CGF07]). However, in cases where the order of tables needs to be maintained, e.g. for queries using merge joins, these solutions still need a CPU-intense key-based `MergeUnion`, and require scanning of all keys in the queries. As such, we exclude these techniques from our evaluation.

6.4 PDT in Detail

In this section, we first list relevant properties and design assumptions about the PDT data structure. We then provide basic PDT algorithms for lookup, merge during scans, and addition of new updates.

6.4.1 Properties

The PDT is a B^+-tree like tree that holds two non-unique monotonically increasing keys, SID and RID, where RID is never explicitly stored, but can always be computed as $SID + \Delta$. Neither SID nor RID are sufficient to uniquely identify a tuple by themselves. Their combination, (SID, RID), is:

Theorem 1. *(SID, RID) is a unique key of a table.*

Proof. We prove by contradiction. Assume we have two tuples with equal SID. The first of these is always a newly inserted tuple, which increments Δ by one. Given that $RID = SID + \Delta$, the second tuple cannot have an equal RID. Next, assume we have two tuples with equal RID. The first of these is always a deleted tuple, which decrements Δ by one. The second tuple can never have an equal SID, as $SID = RID - \Delta$. □

Corollary 2. *If updates within a PDT are ordered on (SID, RID), they are also ordered on SID and on RID.*

Corollary 3. *Within a PDT, a chain of N updates with equal SID is always a sequence of $N - 1$ inserts, followed by either another insert, or a modification or deletion of an underlying stable tuple.*

Corollary 4. *Within a PDT, a chain of N updates with equal RID is always a sequence of $N-1$ deletions, followed by either another deletion, or a modification of the subsequent underlying stable tuple, or a newly inserted tuple.*

6.4.2 Design Decisions

Modify Chains

There is one subtle point that should be made regarding the properties from Section 6.4.1. The combination of (SID, RID) is a key that identifies *entire*

tuples, being them visible or ghost. Strictly speaking, however, it is *not* a key that uniquely identifies update entries within the leaves of our PDT structure. The reason is that a PDT might contain more than one modify update for a single (stable) tuple, one per modified attribute. Such modifies share the same (SID, RID), but can be distinguished by their column-ID, on which we keep them sub-sorted. They do not impact our reasoning about (SID, RID) in any way, as modifies do not contribute to Δ. Furthermore, given that the number of attributes is fixed, worst-case search overhead within a PDT leaf, to locate a certain column-ID, is bound by a constant.

In the preceding and following text, if we talk about "a modification", we generally mean "a tuple with one or more attributes modified". The fact that this is implemented within the PDT as multiple, clustered leaf entries, differentiated by column-ID, is purely a design decision. In the discussion of our algorithms, we often simplify or ignore any details that deal with searching and sorting of same-tuple modifies by column-ID.

Block Deletes

In case a consecutive range of N tuples, starting at SID $= offset$ is deleted, we choose to encode this within the PDT leaves as a single $(DEL, offset, N)$ *block-delete* triplet, rather than N single tuple deletions of the form $(DEL, offset, null)$, $(DEL, offset + 1, null)$, ..., $(DEL, offset + N - 1, null)$. Not only does this reduce PDT memory consumption, but also processing costs for logic that deals with chains of equal-RID deletes (i.e. blocks of consecutive ghost tuples). In the following discussion of our algorithms, however, we stick with the "naive" approach to encode deletes, to avoid cluttering the algorithms with complicating details surrounding block-deletes.

Note that our handling of deletes (including support for block-deletes), differs significantly from what we presented in our original publication of PDTs at SIGMOD 2010 [HZN+10]. Originally, we proposed to not only implement each deletion as a distinct PDT leaf entry, but to also maintain sort-key attribute values for each deleted tuple within the value-space. This allows a very strict tuple ordering to be enforced, where newly inserted tuples are kept sorted even with respect to ghost tuples. With minor modifications to our algorithms, we have been able to loosen this original requirement, allowing us to implement efficient block-deletes and add support for join index updates, at the cost of loosing accuracy in some of our algorithms. We elaborate on this where relevant in the remaining text.

6.4.3 Implementation Details

A memory efficient PDT implementation in C is as follows:

```
typedef struct { uint64 16:n, 48:v } upd_t;
#define F          12          /* fan-out */
#define M          15          /* max leaf entries */
#define INS        65535
#define DEL        65534
#define PDT_MAGIC  0xFFFFFFFFFFFF
#define is_leaf(n) ((n).sid[F-1] != PDT_MAGIC)
#define type(x)    update[x].n /* INS, DEL or ..*/
#define col_no(x)  update[x].n /* column number */
```

```
#define value(x)   update[x].v /* valspace offset or block-delete count */

typedef struct {                typedef struct {
   uint64 sid[M];                  uint64 sid[F];
   upd_t update[M];                sint64 delta[F];
   uint32 prev;                    uint32 child[F];
   uint32 parent;                  uint32 parent;
   uint32 count;                   uint32 count;
   uint32 next;
} PDT_leaf;                     } PDT_intern;
```

This implementation minimizes the leaf memory size, which is important because there will be finite memory for buffering PDT data, thus the PDT memory footprint determines the maximum update throughput that can be sustained in the time window it takes to perform a checkpoint (that allows to free up memory). The leaf of a PDT consists of a SID (large integer), the update type field, and a type-specific value field. The update type has a distinct value for INS,DEL and for each column in the table (for modifies), hence an ultra-wide 65534 column table fits two bytes. The value field occupies the remaining 6 bytes, and either stores an offset into value-space for modify and insert, or the count of consecutive deleted tuples for a block-delete. Overall, each update within the PDT leaves consumes 16 bytes, plus the size of associated data in the value-space and a fractional overhead for a parent pointer and prev/next pointers that link the leaves.

Note that pointers in the tree (child/parent/prev/next) are implemented as integers, rather than physical pointers. These integers represent offsets into a simple buffer manager that manages fixed size blocks of PDT memory, where a single block is used either as a leaf or a node. Since internal nodes need only F-1 SIDs, we can fill the last SID value of internal nodes with a special code that distinguishes them from leaves. The reason that we use logical pointers rather than physical ones, is to allow for easy creation of snapshot copies of a PDT structure (c.f. Section 6.5).

The block-size for PDTs should be aligned with, and a multiple of the cache line size, to optimize memory bandwidth utilization. For example, in case of 64-byte cache-lines, a block size of 256 bytes would accommodate an internal node fan-out of 12 (wasting 8 bytes due to rounding), and would exactly fit a maximum of 15 update entries per leaf.

6.4.4 Lookup by SID and RID

As described in Section 6.2.3, each update is defined with respect to the RID enumerated table image of the target table, as it exists at the start of that update. To insert an update into the PDT, we therefore need to support PDT navigation based on RID. Although RID is not a unique key of the PDT, we do know by Corollary 4 that the currently visible tuple at some RID i is guaranteed to be at the end of an equal-RID chain. Therefore, RID lookups always search for the rightmost leaf containing updates against RID i, which we can achieve using Algorithm 1. The algorithm is a typical tree search, with the main difference that, within an internal node, we can not use binary search over separators (which are SIDs, and we search by RID). We have to perform a linear scan through the SID separators, maintaining a sum of the delta fields for the subtrees that we leave to our left. Using the sum of these deltas, we can compute actual

RID separators on-the-fly, and we follow the child pointer left of the current separator once its value is larger than our target RID. Later on, we will also see scenarios that call for FindLeftLeafByRid, the algorithm of which is left as an exercise for the reader.

Algorithm 1 PDT.FindRightLeafByRid(rid)
PDT.FindLeftLeafByRid(rid)
Finds the rightmost leaf which updates a given rid.
Version that finds the leftmost leaf is omitted.

```
1: node ← rootNode(this)
2: δ ← 0
3: while isLeaf(node) ≡ false do
4:   for i ← 0 to nodeCount(node) do
5:     δ ← δ + node.delta[i]
6:     if rid < node.sids[i] + δ then
7:       δ ← δ − node.delta[i]
8:       break from inner loop
9:     end if
10:  end for
11:  node ← node.child[i]
12: end while
13: return (node, δ)
```

There are also scenarios that call for lookup by SID. For example, indexing in Vectorwise happens at the level of immutable SIDs. To make index lookups "current", we lookup any PDT resident updates against a given SID. This time, the default is to lookup the leftmost leaf, as we might be dealing with an equal-SID chain containing relevant insert data. Algorithm 2 describes the details. Note that finding leftmost leaves is slightly more complex than finding rightmost ones, as SID is not unique and we do not know whether a separator SID cuts an equal-SID chain in two or whether it coincides with the head of such chain. This can be checked in constant time though, and therefore does not impact the logarithmic time complexity of our tree search. Again, there is also a FindRightLeafBySid variant, which we leave out for brevity.

6.4.5 MergeScan

We now move our attention to the Merge operator, which merges a basic Scan on a stable table with the updates encountered during a sequential (left to right) walk through the leaves of a PDT. Algorithm 3 shows the next() method one would typically implement in a relational query processing engine for such an operator; the task of this method is to return the next tuple. The output produced by Merge is an up-to-date, RID-enumerated (strictly increasing over $\mathbb{N}$) tuple stream, which also respects the sort-key ordering of the underlying table.

The idea behind Algorithm 3 is that *skip* represents the distance in position until the next update; until which tuples are just passed through. When an update (INS,DEL,MOD) is reached, action is undertaken to apply the update from the PDT to the output stream. The while loop at line 3 iterates through tuples output by the basic Scan, and either returns them directly as output, or skips them in case they are deleted. If we hit a PDT insert at line 13, we

Algorithm 2 PDT.FindLeftLeafBySid(*sid*)
PDT.FindRightLeafBySid(*sid*)
Finds the leftmost leaf which updates a given *sid*.
Version that finds the rightmost leaf is omitted.

```
1: node ← rootNode(this)
2: δ ← 0
3: while isLeaf(node) ≡ false do
4:    for i ← 0 to nodeCount(node) do
5:       δ ← δ + node.delta[i]
6:       if sid <= node.sids[i] then
7:          δ ← δ − node.delta[i]
8:          if sid ≡ node.sids[i] then
9:             checkLeaf ← true {On equality with SID separator, taking the left
               branch might be too pessimistic}
10:         end if
11:         break from inner loop
12:      end if
13:   end for
14:   node ← node.child[i]
15: end while
16: if checkLeaf ≡ true and not sid ∈ node.sids then
17:    δ ← δ+GetLeafDelta(node)
18:    node ← node.next {Jump to right sibling in case sid not present}
19: end if
20: return (node, δ)
```

retrieve its tuple data from value space and output it, leaving the input Scan untouched. If we end up in the else at line 16, we are left with a modify, which we apply to the current tuple returned by Scan.next(). Note that we may need to process multiple modify entries for the current output tuple, one for each modified attribute.

The listed algorithm is simplified for a single-level PDT. In case of multiple layers of stacked PDTs, the full MergeScan operator consists of a Scan followed by multiple Merge operations, one per PDT layer, where the output of each Merge is used as the Scan input to Merge operation directly above it. Furthermore, the algorithm shown here employs tuple-at-a-time operator pipelining. For our implementation in Vectorwise, which we used for evaluation, we optimized the logic for vectorized execution, where each MergeScan.next() call is expected to return a block of up-to-date tuples. As the *skip* value is typically large, in many cases this allows to pass through entire vectors of tuple data unmodified from the Scan, without copying overhead. For vectors that do get updated, we first analyze the PDT to compute input and output pivots for ranges that can be `memcpy`'d unmodified from input to output. This process leaves space for inserts and compacts out deleted tuples. After this copy phase, we "patch" the output vector with data from inserts and modifies.

6.4.6 Adding Updates

In below discussion of PDT update algorithms, we apply the following simplifications, to stay focused on the core functionality.

Algorithm 3 Merge.next()
The Merge operator has as state the variables $pos, rid, skip$, and DIFF and Scan; resp. a PDT and an input operator. Its next() method returns the next tuple resulting from a merge between Scan and a left-to-right traversal of the leaves of DIFF. pos, rid are initialized to 0, $leaf$ to the leftmost leaf of DIFF and $skip \leftarrow leaf.sid[0]$ (if DIFF is empty, $skip \leftarrow \infty$). Note the new rid is attached to each returned tuple.

```
 1: newrid ← rid
 2: rid ← rid + 1
 3: while skip > 0 or leaf.type[pos] ≡ DEL do
 4:    tuple ←Scan.next()
 5:    if skip > 0 then
 6:       skip ← skip − 1
 7:       return (tuple, newrid)
 8:    else {Delete: do not return the current tuple}
 9:       (leaf, pos) ←DIFF.NextLeafEntry(leaf, pos)
10:       skip ← leaf.sid[pos] − tuple.sid
11:    end if
12: end while
13: if leaf.type[pos] ≡ INS then
14:    tuple ←DIFF.GetInsertSpace(leaf.value[pos])
15:    (leaf, pos) ←DIFF.NextLeafEntry(leaf, pos)
16: else
17:    tuple ←Scan.next()
18:    while leaf.sid[pos] ≡ tuple.sid do {MODs same tuple}
19:       columnId ← leaf.type[pos] {type encodes column number}
20:       tuple[columnId] ←DIFF.GetModifySpace(leaf.values[pos], columnId)
21:       (leaf, pos) ←DIFF.NextLeafEntry(leaf, pos)
22:    end while
23: end if
24: skip ← leaf.sid[pos] − tuple.sid
25: return (tuple, newrid)
```

- We ignore any details around splitting and combining leaf nodes, and growing and shrinking the internal node structure. This logic is similar to a regular B$^+$-tree, with the biggest difference being the subtree delta fields that need to be kept consistent.
- We treat the update entries within the PDT leaves as three dense, aligned arrays: sid[], type[] and value[]. This allows us to ignore navigation among sibling leaves. Also, we ignore boundary cases at both ends of these arrays.
- We do not treat value-space maintenance and garbage collection.
- Handling of block-deletes and modify chains (i.e. multi-attribute modifies of the same tuple) are often ignored. We generalize such modify chains into a single update with type "MOD", whenever being specific about column-IDs does not add any value.

Modify

Algorithm 4 PDT.AddModify($rid, columnId, newValue$)
Finds the rightmost leaf containing updates on a given rid, adding a new modification triplet at index pos, or modify existing PDT data in-place.

```
1: (leaf, δ) ← this.FindRightLeafByRid(rid)
2: (pos, δ) ← this.SearchLeafForRid(leaf, rid, δ)
3: while leaf.sid[pos] + δ ≡ rid and leaf.type[pos] ≡ DEL do {Skip delete-chain}
4:    pos ← pos + 1
5:    δ ← δ − 1
6: end while
7: if leaf.sid[pos] + δ ≡ rid then {In-place update}
8:    offset ← leaf.value[pos]
9:    if leaf.type[pos] ≡ INS then
10:      this.AlterInsertSpace(offset, columnId, newValue)
11:      return
12:   end if
13:   if leaf.type[pos] ≡ columnId then {Modify matches column}
14:      this.AlterModifySpace(offset, columnId, newValue)
15:      return
16:   end if
17:   this.ShiftLeafEntries(leaf, pos, 1) {Add new update triplet to leaf}
18:   leaf.sid[pos] ← rid − δ
19:   leaf.type[pos] ← columnId
20:   offset ← this.AddToModifySpace(columnId, newValue)
21:   leaf.value[pos] ← offset
22: end if
```

To add a modify, we need a row offset, rid, a column identifier, $columnId$, and the new value to be added to the value-space, $newValue$. The column identifier comes from the attribute name in the query plan, while the row offset is the current RID of the tuple being modified, as returned by MergeScan. The first thing we need to do is locate the leaf and the position within that leaf where updates against the tuple with rid should go. This target position may or may not contain existing updates against rid. Within the PDT, a RID looses its uniqueness property, due to ghost-deletes. Therefore, at line 1 of Algorithm 4,

we search for the rightmost leaf that potentially contains updates against our destination RID, to locate the tail of a potential delete-chain. Within that leaf we find the first update entry with a RID greater than or equal to rid using SearchLeafForRid(rid) at line 2. SearchLeafForRid performs a simple left-to-right scan through the PDT leaf, keeping track of the running δ, and comparing rid against the $sid + \delta$ of each update entry encountered. Next, we iterate over a RID-conflicting delete chain, skipping ghost deletes if they exist. Now we are ready to perform the modify at position pos within the leaf, which involves either an in-place update of an existing insert or modify, or the insertion of a new modify triplet in the PDT leaf. To insert a new update entry, we make room within the leaf using ShiftLeafEntries, and add $(SID, columnId, offset)$ triplet, where $offset$ is an index into value-space, holding the new attribute value.

To be complete, when dealing with an in-place modify, we might have to search through a list of attribute modifications for our target tuple, to search for a match on $columnId$ (i.e. the condition at line 13 is simplified). This can even involve searching backwards (i.e. left) through sibling leaves.

Delete

Delete only needs the current RID of the tuple to be deleted. Algorithm 5 adds a deletion to the PDT, and resembles modify in its search for the destination pos to insert at, again skipping any RID-conflicting deletes. If pos happens to already contain an update against rid, we either find an insert, which we simply delete in-place by overwriting it with entries to its right (see line 9), or $\#mod$ modifies against our to-be-deleted rid, which we delete at line 13. Note that after deletion of an insert, we return, while after deleting one or more modifies we continue with the default code path to mark a stable tuple as deleted by adding a DEL entry at pos.

One important difference with respect to modify is the mandatory call to AddNodeDeltas($leaf, val$). This routine adds a (possibly negative) value val to all delta fields of the inner nodes on the path from the root to $leaf$. In case we add a single deletion to our PDT, we increment those node deltas by -1.

Insert

To insert a new tuple into a table, tab, that is sorted according to the set of sort-key attributes, SK, we need to find the position where to insert at. We define this position to be the current RID of the first tuple that should come after our newly inserted tuple in terms of sort-key ordering [4]. To find such insert RIDs, we introduce a merge-based operator, MergeFindInsertRID, to be used in the following way:

```
MergeFindInsertRID(
   Sort(INSERT_BATCH, SK),
   [SK],
   MergeScan(tab, [SK, rid]),
   [SK]
)
```

[4] If no sort-key is defined for the insert table, we simply append to the end, i.e. at a RID that equals the current table count.

Algorithm 5 PDT.AddDelete(rid)
Finds the rightmost leaf containing updates on a given rid. Within that leaf, we either add a new deletion triplet at pos, or delete existing PDT updates in-place.

```
 1: (leaf, δ) ← this.FindRightLeafByRid(rid)
 2: (pos, δ) ← this.SearchLeafForRid(leaf, rid, δ)
 3: while leaf.sid[pos] + δ ≡ rid and leaf.type[pos] ≡ DEL do {Skip delete chain}
 4:    pos ← pos + 1
 5:    δ ← δ − 1
 6: end while
 7: if leaf.sid[pos] + δ ≡ rid then {In-place update}
 8:    if leaf.type[pos] ≡ INS then {Delete existing insert}
 9:       this.ShiftLeafEntries(leaf, pos + 1, −1)
10:       this.AddNodeDeltas(leaf, −1)
11:       return
12:    else {Delete #mod existing modifies}
13:       this.ShiftLeafEntries(leaf, pos + #mod + 1, −#mod)
14:    end if
15: end if
16: this.ShiftLeafEntries(leaf, pos, 1)
17: leaf.sid[pos] ← rid − δ
18: leaf.type[pos] ← DEL
19: leaf.value[pos] ← 0 {Handling of block-deletes left out}
20: this.AddNodeDeltas(leaf, −1)
```

This operator expects as first argument a batch of tuples to be inserted (i.e. a relation), ordered according to the sort-key attributes of the table. The second argument specifies an attribute list identifying the attributes within INSERT_BATCH that represent the sort-key, to be used as attributes for merging. The MergeScan produces an ordered, RID-enumerated scan of the current destination table. Within the output of MergeScan, the attributes to be used as sort-key for merging are identified by the fourth argument. As output, MergeFindInsertRID produces the tuples from INSERT_BATCH, enumerated with their destination RIDs. This RID-enumerated tuple stream can be fed directly into the Insert() operator, which inserts the tuples into the PDT using Algorithm 6.

Note that to find the insert-RIDs, it is sufficient to restrict MergeScan to the SK attribute columns only. Using Vectorwise's MinMax indices, we can further trim down the I/O volume to only contain the horizontal slice that contains the minimum and maximum SK values within INSERT_BATCH. Maintenance of MinMax indices under updates is discussed in more detail in Chapter 7.

Algorithm 6 is less complicated than modify and delete, as there is no need to deal with any in-place updates. As before, we search for the leaf to insert at and the destination position within that leaf, skipping any RID-conflicting ghost tuples. We then simply add the tuple data to the value-space and store a $(SID, INS, offset)$ triplet at destination pos within the PDT leaf, shifting whatever updates might be there to the right. We call AddNodeDeltas($leaf, 1$) to propagate our contributed delta of $+1$ to the delta fields of internal nodes along the root-to-leaf path.

Algorithm 6 PDT.AddInsert($rid, tuple$)
Finds the leaf where updates on rid should go. Within that leaf, we add a new insert triplet at index pos.

1: $(leaf, \delta) \leftarrow this$.FindRightLeafByRid($rid$)
2: $(pos, \delta) \leftarrow this$.SearchLeafForRid($leaf, rid, \delta$)
3: **while** $leaf.sid[pos] + \delta \equiv rid$ **and** $leaf.type[pos] \equiv$ DEL **do** {Skip delete chain}
4: $pos \leftarrow pos + 1$
5: $\delta \leftarrow \delta - 1$
6: **end while**
7: $this$.ShiftLeafEntries($leaf, pos, 1$) {Insert update triplet in leaf}
8: $leaf.sid[pos] = rid - \delta$
9: $leaf.type[pos]$ =INS
10: $offset = this$.AddToInsertSpace($tuple$)
11: $leaf.value[pos] = offset$
12: $this$.AddNodeDeltas($leaf, 1$) {Increment deltas on root-to-leaf path by 1}

SID	col	RID
0	Alpha	0
1	Charlie	1
2	Echo	2

(a) Initial insdel table.

```
INSERT INTO insdel
  VALUES ('Bravo')
DELETE FROM insdel
  WHERE col = 'Charlie'
```

(b) Update statements.

SID	col	RID
0	Alpha	0
1	***Bravo***	1
1	Charlie	2
2	Echo	2

(c) Insert before delete.

SID	col	RID
0	Alpha	0
1	Charlie	1
2	***Bravo***	1
2	Echo	2

(d) Insert after delete violates ordering.

Figure 6.5: Disrespecting deletes.

6.4.7 Disrespecting Deletes

An important subtlety in the design of our PDT structure stems from the interaction between deleted stable tuples, that become "ghost" tuples, and newly inserted tuples. As discussed, a RID is only unique within the up-to-date table image, not within the PDT, as for every *visible* tuple, there might be one or more *ghost* tuples with an equal RID. By design, all our algorithms add updates (by RID) to the *end* of any RID-conflicting delete chain. A consequence of this is that PDT inserts might violate sort-key ordering with respect to RID-conflicting ghost tuples.

An example of this behavior can be found in Figure 6.5, where we see an initial table (6.5a), sorted on col and two update statements, an insert and a delete (6.5b). Although the update statements are independent, the order in which we execute them influences the state of updates in the PDT. If we first perform the insert of 'Bravo', this tuple is inserted after 'Alpha' and before 'Charlie', claiming the current RID=1 of 'Charlie', and sharing its SID=1. Subsequent deletion of tuple 'Charlie', at a shifted RID of 2, results in the state depicted in Figure 6.5c.

If we reverse the execution order of our update statements, we first delete 'Charlie' at RID=1. After this deletion, however, the newly inserted 'Bravo'

does not get inserted right before 'Charlie', but directly before 'Echo', which has become the first *visible* tuple with a sort-key greater than 'Bravo'. The insert still goes to RID=1, which is now shared with the 'Charlie' ghost tuple. Due to our treatment of RID-conflicting delete-chains in Algorithm 6, however, we insert after the ghost tuple. Therefore, 'Bravo' now receives a SID of 2, rather than 1, resulting in a violation of the ordering property with respect to the 'Charlie' ghost. The reverse situation, where a PDT insert ends up *before* a ghost tuple that precedes it in terms of sort-key ordering, is not possible.

The design decision to disrespect deletes mainly impacts index lookups. As discussed in Section 4.4.4, Vectorwise's MinMax indices maintain a mapping between attribute value ranges and immutable SID ranges. Disrespecting deletes results in this mapping to become "loose", as newly inserted attribute values can now cross partition boundaries, resulting in loss of index accuracy over time. In our original PDT paper [HZN+10], we avoided this complication by enforcing strict ordering of inserts with respect to ghost tuples, which required sort-key attributes to be stored in the PDT for each deleted tuple. This strategy turned out to be unworkable with respect to join index updates, as we discuss in Chapter 7, which is the reason we abandoned the strict ordering requirement. An added benefit is that PDTs become more space efficient, as ghost tuples do not claim any value space storage, and delete entries for adjacent tuples can be coalesced into a single block-delete within PDT leaves.

6.5 Transaction Processing

Stacked PDTs can be used as a building block in lock-free transaction processing, as depicted earlier in the architecture from Figure 4.13. The goal here is to provide snapshot isolation, where each new query sees an isolated snapshot of a table, through careful versioning of PDTs. While database systems that offer snapshot isolation as the highest consistency level (e.g. Oracle) are very popular, they still fail to provide full serializability. However, recent research [CRF08] shows that it is possible to guarantee full serializability in a snapshot-based system using extra bookkeeping of read tuples.

To implement snapshot isolation, we propose to use three layers of consecutive PDTs per table. From bottom to top, we have:

read-PDT A global and shared PDT that contains updates with respect to the stable table on disk, i.e. $\text{DIFF}_{t_r}^{t_0}$ with respect to TABLE_{t_0}. This is where the majority of updates resides, making this the largest PDT in general. The read-PDT is considered "read-mostly", hence its name.

write-PDT Another global PDT, containing updates against the merge of a stable table with its read-PDT, i.e. a $\text{DIFF}_{t_w}^{t_r}$ with respect to TABLE_{t_r} = TABLE_0.Merge($\text{DIFF}_{t_r}^{t_0}$). The SIDs in a write-PDT therefore refer to RIDs produced by a merge, implying that the read-PDT can not be arbitrarily modified. In general, modifying any of the global PDTs requires a snapshot copy to be made (copy-on-write), discussed in more detail below.

trans-PDT A transaction-private PDT that contains updates against a merge of the stable table with both the read- and write-PDT, i.e. a $\text{DIFF}_{t_t}^{t_w}$ with respect to TABLE_{t_w} = TABLE_0.Merge($\text{DIFF}_{t_r}^{t_0}$).Merge($\text{DIFF}_{t_w}^{t_r}$). The

trans-PDT ensures that changes made by an update query within a multi-query transaction are visible to subsequent queries within that transaction, while at the same time providing isolation of those changes from other concurrently running transactions. It can be freely modified by its containing transaction, until its updates are either committed into (a copy of) the write-PDT, or simply destroyed in case the transaction aborts.

In the remaining text, we often abbreviate those to RPDT, WPDT and TPDT, or even R, W and T, respectively. The current, transaction-local table image can then be generated with:

$$\text{TABLE}_t = \text{TABLE}_0.\text{Merge}(\text{RPDT}_r^0).\text{Merge}(\text{WPDT}_w^r).\text{Merge}(\text{TPDT}_t^w) \quad (6.8)$$

At the start of a new transaction, x, it receives an empty trans-PDT, Tx, and reference-counted pointers to the current master copies of both the read- and write-PDT, R and W. R and W are guaranteed not to change during x's lifetime, providing a low-cost snapshot of table state at start(x). For read-only transactions, this is all there is. Write transactions, on the other hand, make changes to the private trans-PDT, Tx. [5] These can either be aborted or committed. In case of an abort, we simply destroy Tx. In case of successful commit, the changes in Tx need to be *propagated* into W. However, we can not modify the global write-PDT in place, as that would break isolation. Therefore, we commit Tx's changes into a *snapshot copy*, Wx, of the master write-PDT at the time Tx commits (we treat concurrent commits in Section 6.5.2). When this is done, we update the global W to reference the updated copy, Wx, making it the new master. Concurrent transactions keep seeing the outdated write-PDTs though, as the reference counting keeps previous versions alive as long as needed. Using this "copy-on-write" policy, we ensure that we only pay the cost of creating a snapshot copy when dealing with write transactions that actually commit.

From a perspective of correctness, adding the write-PDT layer is superfluous, as a single global PDT in combination with a trans-PDT is technically sufficient. Architecturally, the motivation for a separate write-PDT is that it can be kept CPU cache resident, while the read-PDT is RAM resident, with the stable table being disk resident. This size advantage of the write-PDT with respect to the read-PDT keeps the cost of creating snapshot copies during commit relatively low. Only when the write-PDT grows too large, we empty it by migrating the updates it contains into a snapshot copy of the read-PDT, amortizing the high cost of such a copy over thousands or even millions of updates. Reducing the number of read-PDT snapshots is not only beneficial in terms of CPU overhead, but also reduces PDT memory consumption, as concurrent transactions do not end up each with distinct references to outdated read-PDT copies. As long as the read-PDT is not changed, which would trigger copy-on-write, all transactions share a reference to the master copy.

Next, we discuss the propagation of updates between PDT layers in more detail. After that, we move to concurrency and conflict resolution, respectively in Sections 6.5.2 and 6.5.3.

[5] A fourth level "query-PDT" can be used during the execution of insert and delete statements, to protect them from seeing their own changes. When such a query finishes, its query-PDT is propagated to its trans-PDT and removed.

Algorithm 7 PDT.Propagate(T)
Propagates the updates present in argument PDT T to this PDT W. It assumes that T is consecutive to W.

```
1: leaf ← T.FindLeftLeafBySid(0)
2: pos ← δ ← 0
3: while leaf do {Iterate over input updates}
4:    rid ← leaf.sid[pos] + δ
5:    if leaf.type[pos] ≡ INS then {Insert}
6:       tuple ← T.GetInsertSpace(leaf.value[pos])
7:       W.AddInsert(rid, tuple)
8:       δ ← δ + 1
9:    else if leaf.type[pos] ≡ DEL then {Delete}
10:      W.AddDelete(rid)
11:      δ ← δ − 1
12:   else {Modify}
13:      columnId ← leaf.type[pos]
14:      val ← T.GetModifySpace(leaf.value[pos], columnId)
15:      W.AddModify(rid, columnId, val)
16:   end if
17:   (leaf, pos) ← T.NextLeafEntry(leaf, pos)
18: end while
```

6.5.1 Propagate

Algorithm 7 lists the Propagate operator that adds to PDT W holding updates from $[t_0, t_1\rangle$, all updates of a consecutive PDT T of time range $[t_1, t_2\rangle$:

$$\mathrm{W}_{t_2}^{t_0} \leftarrow \mathrm{W}_{t_1}^{t_0}.\mathrm{Propagate}(\mathrm{T}_{t_2}^{t_1}) \tag{6.9}$$

with $\mathrm{TABLE}_0.\mathrm{Merge}(\mathrm{W}_{t_2}^{t_0}) = \mathrm{TABLE}_0.\mathrm{Merge}(\mathrm{W}_{t_1}^{t_0}).\mathrm{Merge}(T_{t_2}^{t_1})$.

This operation is used both to commit updates from a trans-PDT into a snapshot copy of the write-PDT and to periodically migrate the contents of the write-PDT into a copy of the read-PDT, to prevent the write-PDT from growing too large. The Propagate algorithm takes all updates in a higher-layer PDT T in left-to-right leaf order and applies them to the PDT layer directly below it, W. Note that the algorithm treats SIDs in T as RIDs of W, keeping track of δ to adjust for preceding updates that already got propagated. This requires the updates in T to be consecutive (see Section 6.2.6) to the updates in W. When propagating from the write- to the read-PDT, we can assume this to be true. However, in case of trans- to write-PDT propagation, this assumption may not hold, because concurrent transactions might have committed, thereby creating a new master write-PDT. Hence, the time ranges represented by the master write-PDT may *overlap* with the trans-PDT of the committing transaction.

6.5.2 Overlapping Transactions

At the start of a transaction x at time t_0, an empty trans-PDT Tx^{t_0} is created. Until x commits at t_2, updates are added to the trans-PDT denoted at commit time $Tx_{t_2}^{t_0}$. If no transaction committed in the meantime, we may just use Algorithm 7 to propagate the updates directly to the master write-PDT: $W_{t_2} = W_{t_0}.\mathrm{Propagate}(Tx_{t_2}^{t_0})$ to get a database state at time t_2 that reflects x. However, if a transaction y committed at t_1, where $t_0 < t_1 < t_2$: we need to do two things:

(1) In snapshot isolation, a conflict occurs if the write-sets of both transactions overlap, so we must check for updates in trans-PDTs Tx and Ty that modify the same tuple.
(2) If no conflicts exist, we need to propagate the changes from Tx into the current master write-PDT W_{t_1}.

To perform (1), the updates in Tx and Ty should be *aligned*, i.e. relative to the *same* database snapshot (matching SID domains). To perform (2), updates in Tx should be *consecutive* with respect to the updates in Ty, i.e. relative to the database snapshot as *produced* by merging updates committed by Ty (SID domain of Tx matches RID domain of Ty). Assuming for now that x and y are the only concurrent transactions, they are guaranteed to have started using the same initial write-PDT (i.e. no other transaction can have committed in the meantime). This makes Tx and Ty aligned. We can now check for conflicting updates in Tx and Ty by analyzing the update SIDs in both PDTs (in ordered and synchronized fashion). While we are checking for conflicts, we can also convert the SIDs of the updates in Tx (the committing transaction) so that they become relative to the RID domain as produced by transaction y, thereby serializing x and y. If Ty is a PDT_{t1}^{t0}, serializing means that we are transforming PDT Tx_{t2}^{t0} into a PDT $T'x_{t2}^{t1}$, which allows us to perform (2).

6.5.3 Serialize

Algorithm 8 lists the Serialize() routine that performs this transformation, but also raises an error (returns false) if there are conflicting updates that make the transposition illegal. In case of such a conflict, the committing transaction must be aborted. Note that the conflict checking performed is write-write on the tuple-level and even allows to reconcile modifications of different attributes of the same tuple (by the omitted CheckModConflict() routine).

$$T'x_{t_2}^{t_1} \leftarrow Tx_{t_2}^{t_0}.\text{Serialize}(Ty_{t_1}^{t_0}) \tag{6.10}$$

Algorithm 8 iterates through the leaves of both the committing Tx and the committed Ty, from left to right, in lock-step fashion. All update entries in Tx have their SID "transposed" from an aligned into a consecutive state by adjusting those SIDs with the running delta from Ty, thereby shifting the SID positions in Tx to accommodate for any updates in Ty that precede it. Whenever matching SIDs are encountered in Tx and Ty, we check for conflicts, returning false whenever Tx tries to modify or delete a tuple that was already deleted by Ty, or when Tx tries to delete or modify a tuple/attribute modified by Ty. Inserts only conflict in case of a key uniqueness violation, otherwise we simply resolve the conflict by ordering the tuples according to sort key.

6.5.4 Commit

Until now we only considered two concurrent transactions, but Algorithm 9 extends the idea to an arbitrary number of concurrent transactions.

For each recently committed transaction z_i that overlaps with a still running transaction x we keep their *serialized* trans-PDTs $T'z_i$ alive in the set TZ. A reference counting mechanism ensures that $T'z_i$-s are removed from TZ as soon as the last overlapping transaction finishes. Basically, all $T'z_i$ are consecutive

Algorithm 8 PDT.Serialize(Ty)
This method is invoked on PDT Tx with an aligned Ty as input, and checks the updates in Tx (the newer PDT) for conflicts with an earlier committed transaction Ty. FALSE is returned if there was a conflict. Its effect on Tx (referred to as $T'x$) is to become consecutive to Ty, as its SIDs get converted to the RID domain of Ty.

```
1: imax ← Tx.count()
2: jmax ← Ty.count()
3: i ← j ← δ ← 0
4: while i < imax do {Iterate over new updates}
5:   while j < jmax and Ty[j].sid < Tx[i].sid do
6:     if Ty[j].type ≡ INS then
7:       δ ← δ + 1
8:     else if Ty[j].type ≡ DEL then
9:       δ ← δ − 1
10:    end if
11:    j ← j + 1
12:  end while
13:  if Ty[j].sid ≡ Tx[i].sid then {potential conflict}
14:    if Ty[j].type ≡ INS and Tx[i].type ≡ INS then {INS-INS}
15:      if Ty[j].value < Tx[i].value then {old value goes before}
16:        δ ← δ + 1
17:        j ← j + 1
18:      else if Ty[j].value ≡ Tx[i].value then {key conflict}
19:        return false
20:      else {new value goes before}
21:        Tx[i].sid ← Tx[i].sid + δ
22:        i ← i + 1
23:      end if
24:    else if Ty[j].type ≡ DEL then
25:      if Tx[i].type ≢ INS then
26:        return false
27:      else {Never conflict with Insert}
28:        Tx[i].sid ← Tx[i].sid + δ
29:        δ ← δ − 1
30:        i ← i + 1
31:      end if
32:    else {Modify in Ty}
33:      if Tx[i].type ≡ INS then {Insert in Tx goes before modify in Ty}
34:        Tx[i].sid ← Tx[i].sid + δ
35:        i ← i + 1
36:      else if Tx[i].type ≡ DEL then {DEL-MOD conflict}
37:        return false
38:      else if CheckModConflict() then {MOD-MOD}
39:        return false
40:      end if
41:    end if
42:  else {Current SID in Ty is bigger than in Tx}
43:    Tx[i].sid ← Tx[i].sid + δ {Only convert SID}
44:    i ← i + 1
45:  end if
46: end while
47: return true
```

Algorithm 9 Finish(ok, W_{t_n}, Tx^t, TZ)
Commit(w, tx, tz) = Finish(true,w, tx, tz)
Abort(w, tx, tz) = Finish(false,w, tx, tz)
Transaction x that started at t, tries to commit its trans-PDT Tx^t into master write-PDT W_{t_n}, taking into account the sequence of relevant previously committed consecutive PDTs $TZ = (T'z1_{t_1}^{t_0}, \cdots, T'zn_{t_n}^{t_{n-1}})$. If there are conflicts between Tx and any $T'zi \in TZ$, the operation fails and x aborts. Otherwise the final $T'x$ is added to TZ and is propagated to W_{t_n}.

1: $T'x \leftarrow Tx; i \leftarrow 0$
2: **while** $(i = i + 1) \leq n$ **do** {iterate over all $T'zi$}
3: $T \leftarrow T'zi_{t_i}^{t_i - 1}$
4: **if** $t < t_i$ **then** {overlapping transactions}
5: **if** ok **then**
6: $ok \leftarrow T'x$.Serialize(T)
7: **end if**
8: $T.refcnt \leftarrow T.refcnt - 1$
9: **if** $T.refcnt \equiv 0$ **then** {x is last overlap with zi}
10: $TZ \leftarrow TZ - T$
11: **end if**
12: **end if**
13: **end while**
14: **if** $ok \equiv$ **true then** {only commit x on success}
15: $W_{t_{n+1}} \leftarrow W_{t_n}$.Propagate($T'x$)
16: $T'x.refcnt \leftarrow \|runningtransactions\|$
17: **if** $T'x.refcnt > 0$ **then**
18: $TZ \leftarrow TZ + T'x$
19: **end if**
20: **end if**
21: **return** ok

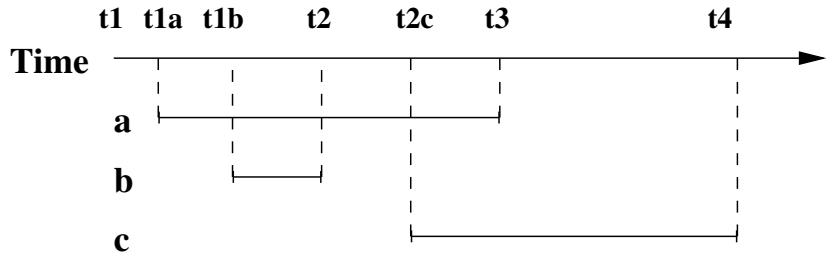


Figure 6.6: Three Concurrent Transactions

and hold the changes that the transaction z_i applied to the previous database state. The creation of such a $T'z_i$ is in fact a by-product of the fact that z_i committed. Like described in the two-transaction case, committing entails using the Serialize algorithm to transform the trans-PDT Tx of the committing transaction possibly multiple times; once for each overlapping transaction $z_i \in TZ$ in commit order. The execution of Serialize both serves the purpose of checking for a write-write conflict (which leads to an abort of x), as well as produces a serialized trans-PDT $T'x$ that is consecutive to the database state at commit time, hence can be included in TZ and also used to Propagate its updates to the master write-PDT (i.e. commit).

6.5.5 Example

The example in Figure 6.6 shows concurrent execution times of three transactions: a, b, and c. At start time t_1, we start out with an initial master write-PDT $W_{t_1} = \emptyset$. At t_{1a}, the first transaction, a, arrives and sees the current master write-PDT: $Wa = W_{t_1} = \emptyset$. Here, we chose notation t_{1a} to suggest that the database state at that time is the same as at t_1. Transaction a starts out with an empty trans-PDT, $Ta^{t_{1a}} = \emptyset$. At t_{1b}, the second transaction, b, arrives and sees the same empty write-PDT, $Wb = Wa$, since no commits took place in the meantime. At t_2, transaction b commits, thereby propagating its changes from $Tb_{t_2}^{t_{1b}}$ to the current table image. There were no commits during the lifetime of b, so the most up-to-date table image is still represented by $Wb = W_{t_1}$, allowing us to commit by propagating Tb directly into a snapshot copy W'_{t_1} of W_{t_1} and making the result the new master write-PDT: $W_{t_2} = W'_{t_1}$.Propagate($Tb_{t_2}^{t_{1b}}$). Note that copy-on-write, together with reference counting, make sure that transaction a still sees the outdated $Wa = W_{t_1} = \emptyset$. The resulting new master write-PDT reflects the state as seen by incoming transaction c at t_{2c}: $Wc = W_{t_2}$. Again we start with an empty $Tc^{t_{2c}} = \emptyset$ for transaction c. The next thing that happens is the commit of transaction a at t_3. We serialize $Ta_{t_3}^{t_{1a}}$ with respect to $Tb_{t_2}^{t_{1b}}$: $T'a_{t_3}^{t_2} = Ta_{t_3}^{t_{1a}}$.Serialize($Tb_{t_2}^{t_{1b}}$), which reports no conflicts. The resulting $T'a_{t_3}^{t_2}$ is consecutive to W_{t_2}, a snapshot-copy of which we can safely commit into: $W_{t_3} = W'_{t_2}$.Propagate($T'a_{t_3}^{t_2}$). Finally, when transaction c commits, at t_4, we still have $T'a_{t_3}^{t_2}$ around, which is aligned with $Tc^{t_{2c}}$, so we can do $T'c_{t_4}^{t_3} = Tc_{t_4}^{t_{2c}}$.Serialize($T'a_{t_3}^{t_2}$), which can then be propagated into the write-PDT: $W_{t_4} = W'_{t_3}$.Propagate($T'c_{t_4}^{t_3}$).

In summary, at transaction commit, we check for conflicts against all transactions that committed during the lifetime of the transaction. Each such over-

lapped commit is characterized by its serialized trans-PDT; we cache these for those recent transactions that still overlap with a running transaction. By serializing the committing trans-PDT with these cached PDTs one-by-one, we finally obtain a trans-PDT that is consecutive to the current database state, and can be Propagate-ed to it (and added to the cache itself). These Serialize operations will also detect any conflicts, in case of which the transaction gets aborted instead. Together, this amounts to optimistic concurrency control.

6.6 Logging and Checkpointing

All PDT operations discussed thus far operate on memory resident data structures. However, we still need to access disk on two occasions. First, to guarantee durability of committed updates, contents of each committing trans-PDT are written to a *write-ahead-log* (WAL). Second, to guarantee availability and performance over the long run, memory resident PDT updates need to be written to their respective stable tables on disk periodically. We do this by creating a new version of a stable table, including its indexing structures, using a *checkpoint* operation. Both these operations involve purely sequential I/O.

To append a trans-PDT to the WAL, during commit, we flush the update entries in its leaves, sequentially and densely packed, together with the attribute values from its value space, to the tail of the log. If writing the complete log entry fails, the transaction fails to commit, thereby guaranteeing atomicity. During a restart of the database server, we walk through the WAL from start to end, and gather the updates for each table from the log into a big initial read-PDT, using an algorithm akin to Propagate on the update entries we encounter. In effect, this replays the trans-PDT commits in their original order. If an incomplete trans-PDT entry is found, which is only possible at the tail of the log, this indicates a system failure, and this final trans-PDT is ignored to ensure consistency.

To free up PDT memory and reduce merging overhead, both of which increase with the number of updates in a tables PDTs, we use the checkpoint operation to build a new stable table that includes all committed updates from the global read- and write-PDTs. To checkpoint a stable table, TABLE_0, into a new version, TABLE'_0, we start a background checkpointing transaction, illustrated in Figure 6.7, which proceeds as follows.

1. Flush any updates from the write-PDT to the read-PDT using Propagate.

2. Temporarily disable write-to-read propagation, thereby locking down the read-PDT to prevent its modification during the checkpoint.

3. Build the new TABLE'_0 by performing a MergeScan of TABLE_0 with the locked-down read-PDT (and the empty write-PDT), appending its output to the storage of a freshly instantiated TABLE'_0, i.e. Append(TABLE'_0, MergeScan(TABLE_0)). Append takes care of automatically rebuilding MinMax indices and recompressing the data when applicable.

4. Switch to the new table image by updating the catalog entry for the checkpointed table.

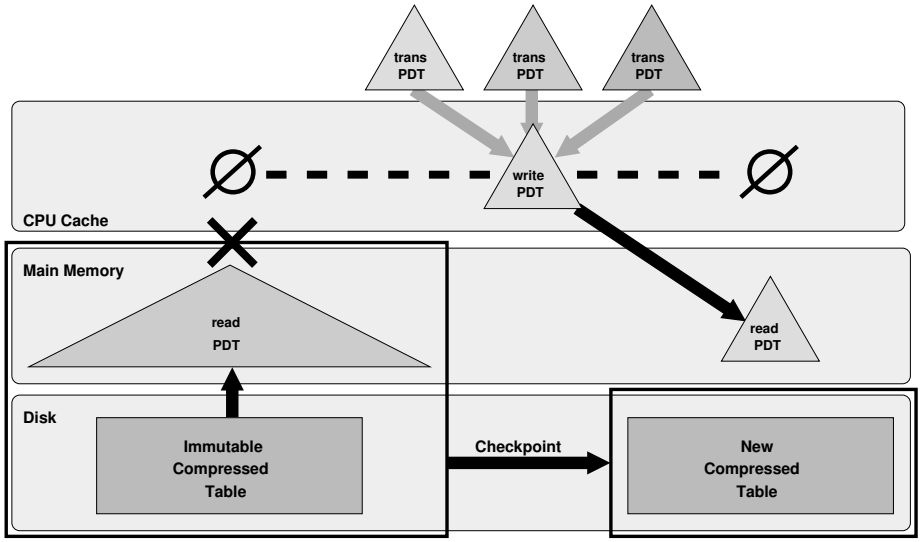


Figure 6.7: Rebuilding a stable table with a Checkpoint.

5. Destroy the old table image and its read-PDT to free up their storage and memory, respectively. [6]

6. Set the current write-PDT, which contains all updates committed during the lifetime of the background checkpoint, to be the initial read-PDT of the new table, adding an empty write-PDT above it.

7. Remove all trans-PDT entries for the checkpointed table from the WAL.

8. Re-enable write-to-read propagation.

Here we see another advantage of the write-PDT layer, as it allows us to run and commit transactions concurrently with an ongoing checkpoint transaction. Write-PDT updates gathered during a checkpoint, are consecutive to the locked-down read-PDT that got checkpointed. Therefore, these updates are compatible with the SID enumeration of our newly checkpointed TABLE'_0 on disk, allowing us to promote a copy of such a write-PDT into the initial read-PDT of our new table.

A transaction that is still running at the time a checkpoint commits, can safely commit into the new state as well, after performing the necessary serialization against transactions that committed during its lifetime, as discussed in Commit (Algorithm 9). Transactions that start after completion of a checkpoint are guaranteed to see the new image.

For high-volume "bulk" updates, Vectorwise provides a non-SQL-standard *Combine* operator, which allows data from one or more staging tables to be added (i.e. union) or deleted (i.e. difference) to/from an existing base table. This allows large update volumes, which would trigger a checkpoint anyway, to bypass in-memory PDT storage, and simply rebuild a stable table directly. The Combine operation is similar to an automatic checkpoint, but has more

[6] We use reference counting to ensure that after the switch, queries that are still accessing the old table image and read-PDT keep having access to the old state as long as needed.

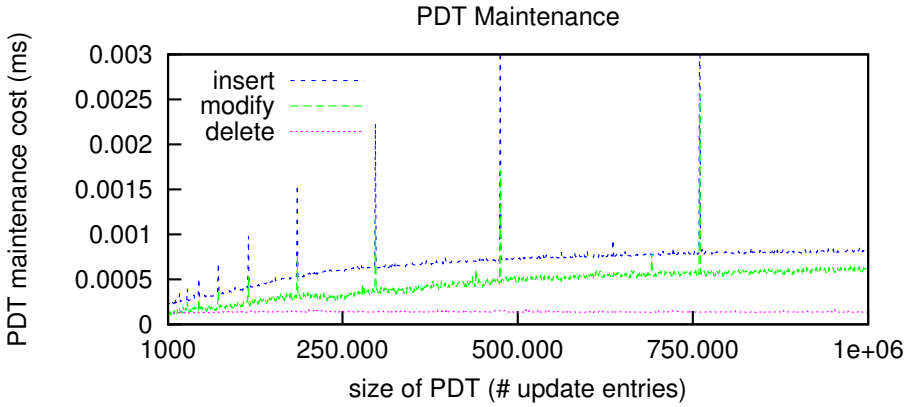


Figure 6.8: PDT update performance over time.

conservative transactional semantics, as it conflicts with concurrent updates to PDTs of given table (i.e. rebuilding a stable table with external data invalidates PDT positions).

6.7 Experimental Evaluation

To evaluate the benefits of the proposed techniques, we run two sets of benchmarks. First, with micro-benchmarks we demonstrate the performance of inserting, deleting and modifying data of varying size using PDTs. We also measure the performance of MergeScan, using different sort key data types and update rates. Then, we investigate if PDTs succeed in providing uncompromising read performance in large-scale analytic query scenarios, using the queries from the TPC-H benchmark.

6.7.1 Benchmark Setup

For our experiments we used two hardware platforms. The workstation system is a 2.40GHz Intel Core2 Quad CPU Q6600 machine with 8GB of RAM and 2 hard disks providing a read speed of 150MB/s. The server system is a 2.8GHz Intel Xeon X5560 machine with 48GB of RAM and 16 SSD drives providing 3GB/s I/O performance. The micro-benchmarks were performed on workstation, were memory-resident and the results are averaged over 10 consecutive runs. The standard deviation over these runs is very small and thus not reported. The TPC-H benchmarks were performed on both workstation and server.

6.7.2 Update Micro Benchmarks

The first set of experiments demonstrates the logarithmic behavior of PDTs as more updates are added to it. Figure 6.8 depicts the average time needed to perform either an insert, a modify, or a delete, into a growing PDT (up to 1 million update entries). Insert and modify costs develop logarithmically, with inserts being slightly more expensive, as the destination table consists of two

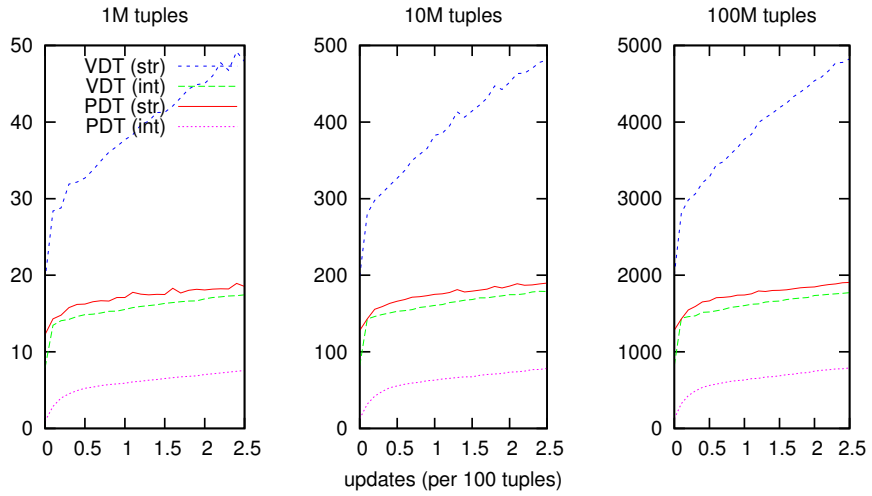


Figure 6.9: MergeScan: Update ratio (as percentage on x-axis) versus scan time (in ms on y-axis), for PDT and VDT with different key types.

(64-bit integer) columns, while modifies involve a single attribute only. The peaks indicate resizing the value space, where attribute values are stored, which is again more expensive for insert than for modify. Resizing does not occur for deletes, as these do not store attribute values. Furthermore, delete stays almost flat, as PDT growth is limited due to the joining of adjacent deletes into a single block delete (we perform 1 million deletions from a 1 million row table).

6.7.3 MergeScan Micro Benchmarks

Figure 6.9 presents the results of scanning a table of 4 columns and 1 key column (integer or string) with updates managed by PDTs and VDTs. The query used is a simple projection of all 4 columns after a varying number of updates have been applied. In all cases PDT outperforms VDT by at least a factor 3. Furthermore, this experiment demonstrates linear scaling of query times with growing data size for both PDT and VDT.

The benefits of PDT are especially visible when the key column contains strings. In that case, as the percentage of updates is increased, value-based merging in VDTs becomes significantly slower due to expensive string comparisons. On the other hand, PDTs do not need to perform value comparisons, thus their cost is lower and does not increase significantly with update ratio.

The next set of experiments investigates the impact of increasing the number of key columns in a table of 6 columns. Here we expect VDTs to suffer when instead of a single-column sort key we have multiple sort columns, as the value-based merge-union and -diff logic becomes significantly more complex with multi-column keys. As in PDTs MergeScans do not need to look at the sort key columns, they are not influenced by this at all. Figure 6.10 depicts the results for both integer and string type of keys. The x-axis is divided into 2 dimensions: for each update percentage we conduct the experiment with a varying number of keys, from 1 to 4 columns. The query projects the remaining

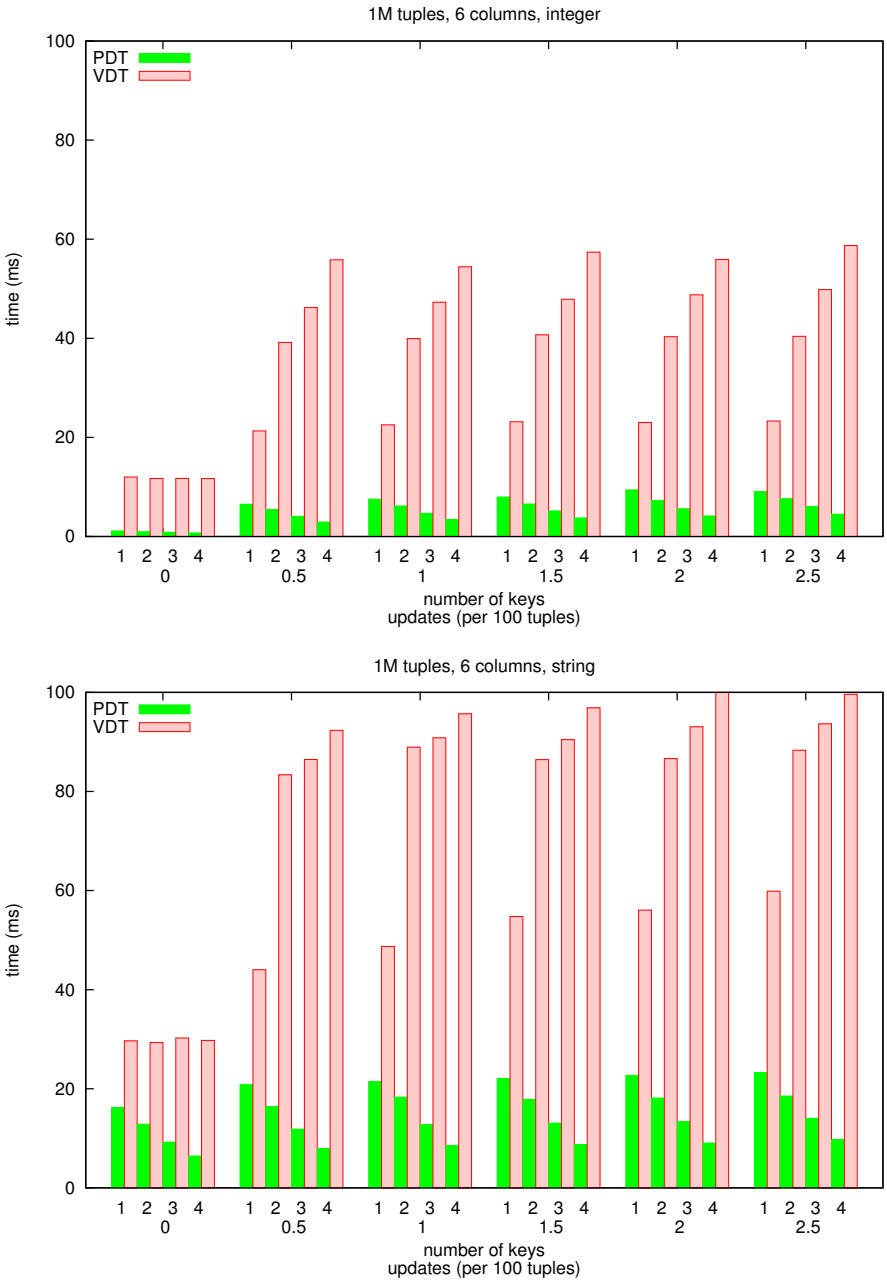


Figure 6.10: MergeScan: scan performance for varying number of sort key attributes, when scanning remaining non sort key attributes.

non-key columns. For VDTs, the query time increases significantly when the number of keys to be scanned and compared is increased. For PDTs, query time decreases because fewer columns have to be projected when the number of keys increase, while merge cost stays constant.

Overall, presented micro-benchmarks demonstrate that PDTs are significantly more efficient than VDTs, especially with complex (string, multi-attribute) keys.

6.7.4 TPC-H Benchmarks

Whereas in the micro-benchmarks we only focused on the PDT operations under controlled circumstances, we now shift our attention to overall performance of analytic queries using the full TPC-H query set (22 queries). The focus in these experiments is establishing whether PDTs indeed succeed in allowing column-stores to be updated, without compromising their good read-only query performance. We compare clean queries (no-updates), that is, queries to a clean database that only has been bulk-loaded, to queries to a database that has been updated by the two official TPC-H update streams, each of which alters (insert and delete) roughly 0.1% of two main tables: lineitem and orders. We test both PDTs and VDTs implemented inside the Vectorwise system on both workstation and server systems described above.

The experiments were conducted for scientific evaluation only, and do not conform with the rules imposed on an official TPC-H benchmark implementation. Therefore, we omit any overall score, and explicitly note that these individual query results should *not* be used to derive TPC-H performance metrics.

The lineitem table in our setting is ordered on a {l_orderkey, l_linenumber} key, while the orders table is ordered on a {o_orderdate, o_orderkey} combination. Due to this ordered columnar table storage, which is very much like row-wise "index-organized" table storage (a clustered index), certain queries can exploit range predicates; however, the update task is non-trivial, as the inserts touch locations scattered throughout the tables.

On the server, we measured the performance using a compressed SF-30 (30GB) data set, while on workstation we used an uncompressed SF-10 (10GB) data set. The results are presented in Figure 6.11 For each of the 3 scenarios a separate bar is plotted for every TPC-H query.[7]

We provide two types of results: *(i)* I/O volume consumed and *(ii)* query performance, separated into data-scanning (including reading from disk, decompression, and applying updates, as applicable), and query processing time. All results are normalized against the VDT runs, with the absolute numbers provided for those.

The top two segments of Figure 6.11 present the results on the server, using a compressed, disk-resident ("cold") SF-30 data set. Plot 2 shows that the I/O volume for VDT runs is consistently higher (or equal) than in no-updates and PDT runs, due to mandatory scanning of sort key columns in VDT merging. However, on this platform, the I/O difference is relatively small, due to good compression ratios for the (sorted) key columns. Still, the overhead of

[7]Queries 2, 11 and 16 do not touch updated tables, hence the results for them do not differ between runs.

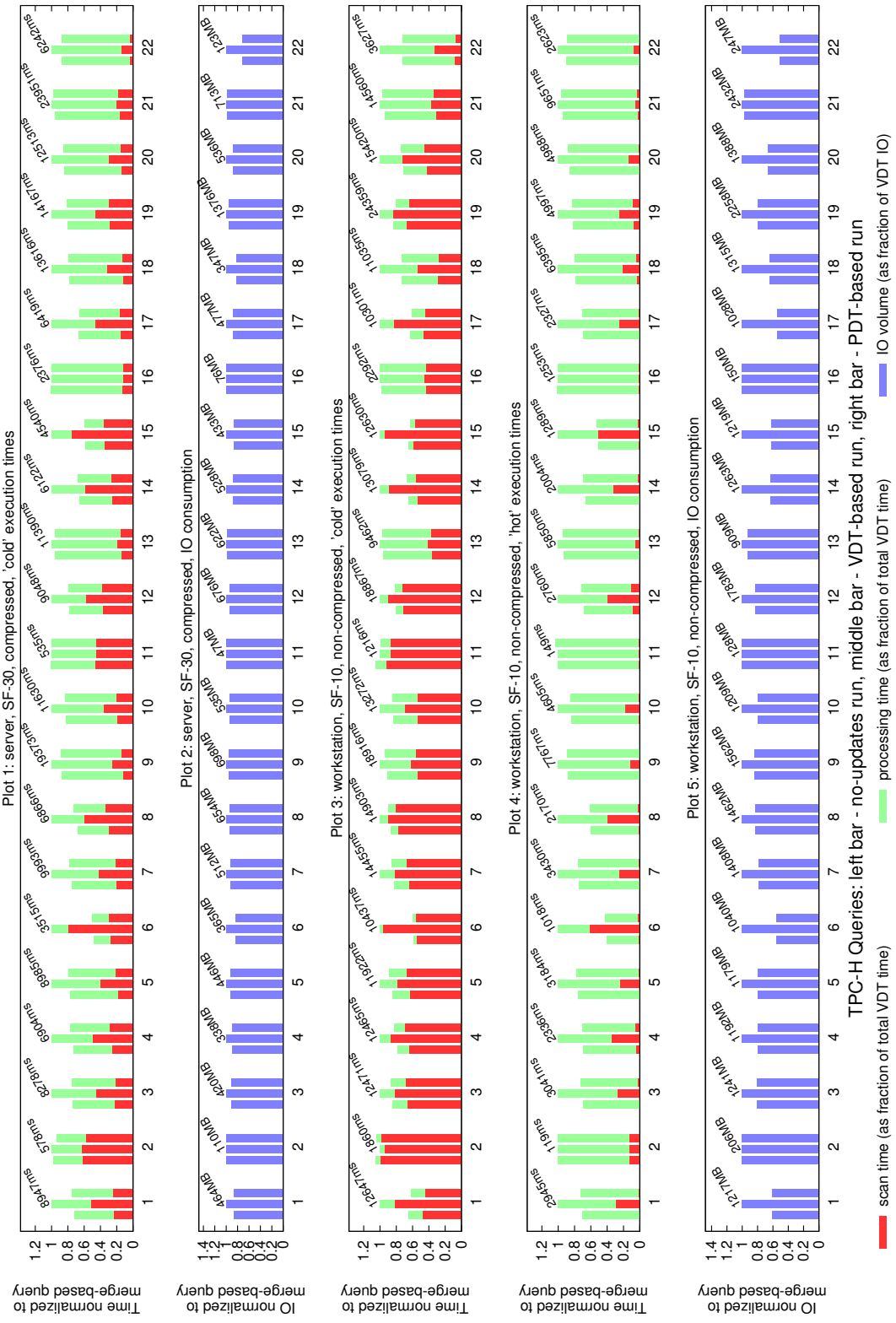


Figure 6.11: TPC-H: server (compressed SF-30) vs workstation (non-compressed SF-10).

value-based merging, visible in Plot 1, can make the "scan" part often significantly higher for the VDT scenario. PDT runs demonstrate a very small "scan" overhead over no-updates, resulting in a negligible impact on the total query time.

The bottom 3 segments of Figure 6.11 present results on the workstation, using a non-compressed SF-10 data set, both memory- ("hot") and disk-resident ("cold"). Plot 5 shows that with non-compressed keys, the I/O volume in the VDT case is significantly higher, up to a factor 2. This demonstrates the benefit of PDTs on tables with keys that have large physical volume (strings; hard to compress; multi-column). This increase in I/O is directly reflected in the performance of the "cold" runs (Plot 3). Plot 4 demonstrates a scenario where lack of compression (which constitutes a significant part of the "scan" cost in Plot 1), combined with memory-resident data, makes accessing data a "zero-cost" operation for the no-updates runs. With the data-access cost eliminated, Plot 4 demonstrates the absolute CPU cost of applying the updates to the data stream. Here, the VDT approach can have a very significant overhead, consuming up to half of the total processing time (e.g. in query 6). The CPU cost for PDT merging is significantly lower, and negligible in most cases.

In all, we can see that value-based merging can be significantly slower (>20%) than positional merging as introduced by PDTs, and PDTs consistently achieve query times very close to querying a clean database.

6.8 Related Work

Differential techniques [SL76] historically lost out to update-in-place strategies as the method of choice to implement database updates handling. Gray pointed out ("UPDATE IN PLACE: A poison apple?") that differential techniques, which do not overwrite data, are more fail-safe, but recognized that magnetic disk technology had seduced system builders into update-in-place by allowing fast partial file writes [Gra81]. Update-in-place also implies random disk writes, where hardware improvement has lagged compared to sequential performance. While this thesis does not touch the subject of using differential techniques for row-based systems, it recognizes differential techniques as the most salient way to go for column-stores, where update-in-place is additionally hindered by having to perform I/O for each column, aggravated by compressed and replicated storage strategies.

The idea to use differential techniques in column-stores is not new (the idea to do so positionally, is) and was suggested early on [CK85]. The Fractured Mirrors approach [RDS03] that combined columnar with row storage, also argued for it. Both papers did not investigate this in detail. The open-source columnar MonetDB system uses a differential update mechanism, outlined in [Bon02]. C-Store proposed to handle updates in a separate write-store (WS), with queries being run against an immutable read-store (RS), and changes merged in from the WS on-the-fly. We see our PDT proposal as particularly suited for column-stores, because our positional approach allows queries that do not use all key columns to avoid reading them, a crucial advantage in a column-store, and merging of updates during a scan is computationally faster, compared to value-based merging.

As for previously proposed differential data structures, the Log-Structured

Merge-Tree (LSMT) [OCGO96] consists of multiple levels of trees, and reduces index maintenance costs in insert/delete-heavy query workloads. Similarly, [JNS+97] proposes multi-level indexing. The goal of improving insertions using not stacked, but *partitioned* B-trees was explored in [Gra03]. A possible reason why multi-tree systems so far have not been very popular, is that lookup requires separate I/Os for each tree. The assumption in our PDT proposal, and similarly in the value-based "VDT" (our terminology) approaches used e.g. in MonetDB, is that only the lowest layer data structure (columnar storage) is disk-resident and requires I/O. The availability of 64-bits systems with large RAM thus plays in its advantage.

Finally, we have shown how PDTs can be used as an alternative way to provide ACID transactions; an area of database functionality where ARIES [MHL+92] is currently the most prominent approach. There are commonalities between ARIES and our proposal, like the use of a WAL and checkpointing, but the approaches differ. ARIES uses immediate updates, creating dirty pages immediately at commit, rather than buffering differences. Instead of tuple-based locking and serializability used in ARIES-based systems, column-stores tend to opt for snapshot isolation, typically with optimistic concurrency control. PDTs fit this approach, offering lock-free query evaluation, using three layers of PDTs (Trans,Write,Read). While snapshot isolation allows anomalies [BBG+95], user acceptance for it is high, and recently it was shown that snapshot-based systems can provide full serializability [CRF08] with only a limited amount of bookkeeping. In all, we think that given hardware trends, and the specific needs of column-stores, PDT-based transaction management is a new and attractive alternative.

6.9 Related Systems

In this section we provide a brief overview and comparison of the update architectures employed by other compressed column-store systems. This is based on papers that were published *after* our original PDT publication.

6.9.1 Microsoft SQL Server

Microsoft recently added compressed columnar storage support to its SQL Server product to enhance performance on analytic workloads [LCH+11, LCF+13]. They also implement vectorized execution to improve processing efficiency. A schematic overview of the storage layout and update support structures is depicted in Figure 6.12.

The bulk of tuples in a table is stored in immutable, compressed, columnar storage format, with no support for explicit tuple ordering. An in-memory *bit map* [OQ97] is employed to mark deleted tuples within the immutable storage. On disk, the bit map is represented and maintained as a B+-tree, indexed by a unique tuple identifier. Deleted tuples are filtered out transparently during a scan.

New tuples can be inserted into a *delta store*, which is a regular B+-tree. There can be multiple such delta stores per table, and all of them are included transparently during a table scan. Direct modification of attribute values (i.e. SQL UPDATE) is not supported, but rather implemented as a deletion of the

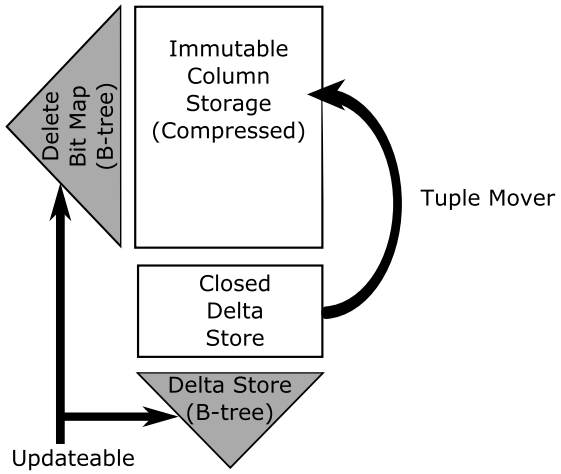


Figure 6.12: Storage and updates in Microsoft SQL Server.

(*entire*) original tuple, followed by reinsertion of the modified version of the tuple. A delta store can be *closed*, which blocks further updates to it, and marks the tuples in that delta store to be ready for migration into immutable storage.

The *tuple mover* looks for closed delta stores, and migrates its tuples into a new compressed *segment* within immutable storage. After moving a delta store, the new segment is made visible, and the delta store invisible, in a single atomic operation. During tuple movement, concurrent scans are allowed, as are inserts into other (non-closed) delta stores. Concurrent deletes (and modifies, being based on deletion), however, are blocked until the tuple mover releases the delete bit map.

The design of SQL Server's update architecture is relatively straightforward, and uses proven components (i.e. B^+-tree's), that were readily available. Although also differential in nature, the architecture differs significantly from Vectorwise's PDT stack. Transactions containing a mix of inserts and deletes (or a modify), need to access at least two global data structures, inducing I/O writes and mutual exclusion mechanisms, which in Vectorwise only occur during commit (briefly) and a checkpoint.

Modify updates are especially expensive in SQL Server, as they require *(i)* a full tuple to be retrieved from columnar storage (I/O proportional to the number of attributes), *(ii)* the original tuple to be marked deleted (leading to wasted I/O bandwidth as the tuple is not physically deleted), and *(iii)* reinsertion of the full (modified) tuple into a row-oriented delta store, which lacks the scan-friendly layout of a compressed column-store. Vectorwise allows modify updates to be added to a PDT on a per-attribute basis, which requires only that attribute to be accessed during the update query. During scans, the old value is still being read, only to be "patched" with the new value, leading to an effect similar to *(ii)*, but *only for the modified attribute*, not the full tuple. Also, Vectorwise does not suffer at all from *(iii)*, as no full tuple is reinserted, and PDT updates are in-memory only (no disk-resident, row-oriented structures to be updated/scanned along).

The lack of support for sorted table storage has pros and cons. The main disadvantage is that query evaluation performance will suffer from a lack of merge-based default plans. Insertion of new tuples, however, becomes relatively straightforward, as no tuple ordering needs to be determined, so that new tuples can be just added to a delta store, and delta tuples may be positioned anywhere in the output of a scan, i.e. no value-based merging is required. Furthermore, each delta store can be migrated independently to a private, compressed column-store segment by the tuple mover. In case of sorted tables, a (partial) table rebuild would be required, in general, to interleave new delta tuples with existing read-store ones. Vectorwise supports both ordered and unordered (i.e. "heap") tables, allowing such trade-offs to be judged by the user.

It is also important to realize that SQL Server's delta tuples are row-oriented. Therefore, as the fraction of delta store tuples in a table increases, especially if it grows beyond memory capacity, scan performance will slowly degrade towards row-store performance. With many deletes (and thus also modifies!) against the immutable data, the situation becomes even worse, as scanned table data is full of useless ghost tuples, wasting precious I/O bandwidth.

The tuple mover is responsible for keeping delta stores from growing large, by migrating them to compressed column storage. This will eventually lead to an increasingly fragmented storage layout. Deletions further fragment every compressed segment, as ghost data is not physically deleted, and the delete bit map can not be migrated to column storage. Both forms of fragmentation can only be resolved by a full table rebuild. In [LCF+13] this is mentioned only briefly, and no further details are provided. If regular tuple movement already hampers SQL Server concurrency, it would be interesting to know the performance implications of a full table rebuild. The checkpointing process of Vectorwise can rebuild both ordered and unordered tables, thereby adding new/modified tuple data and compacting out deletions within read-optimized storage, while still allowing concurrent read and write transactions to start or commit. Vectorwise basically buffers updates in memory until a full table rebuild must be performed, while SQL Server tries to delay a full rebuild by flushing updates to disk more often, thereby accepting *(i)* additional I/O, *(ii)* storage fragmentation, and *(iii)* a lack of sorted storage.

6.9.2 Vertica

Like Vectorwise, Vertica is a commercial DBMS (now owned by Hewlett-Packard (HP)) that was built from scratch to achieve high analytic query performance on modern hardware, and has its roots in academia (i.e. C-store [Sto05]). Vertica also relies on vectorized execution and a compressed, column-oriented storage model to optimize for data- and computation-intensive workloads. Distinguishing features of Vertica are its focus on distributed execution (on a cluster) and the ability to replicate subsets of table columns as a *projection*, according to some interesting sort order (i.e. a column-store alternative to a secondary index). In certain scenarios, Vertica also supports query evaluation on compressed data, i.e. without decompressing it. What follows is a brief overview of Vertica and a comparison against Vectorwise. This discussion is fully based on information provided in [LFV+12].

In Vertica, the bulk table data is stored on disk in what is called *read optimized storage* (ROS), containing compressed columnar data, while incoming

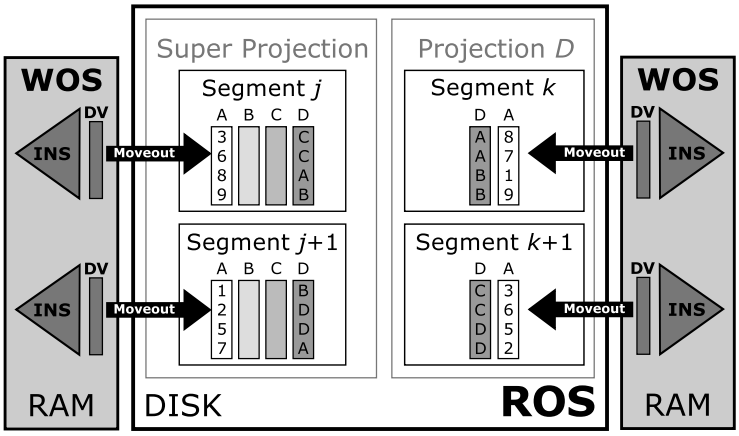


Figure 6.13: High-level storage layout of a single node in Vertica. The figure shows a simple example table, stored in its entirety in the *super projection*, sorted on A, and partially replicated in a projection that only contains attributes A and D and is sorted on D. Both projections are partitioned into *segments*. Each projection may have its own segmentation expression, but for simplicity we ignore this fact and the expression itself. Inserts (INS) and deletions, in a *delete vector* (DV), in memory-resident WOS also respect this segmentation. The tuple mover is responsible for migrating tuples from WOS to ROS through a process called *Moveout*.

updates are added to *write optimized storage* (WOS), which is fully in-memory and row-oriented. In ROS, *all* columns of a table must at least be represented in what is called the *super projection* of the table. Besides this super projection, each table may be replicated into zero or more regular *projections*, each with a distinct sort order and subset of table attributes. Each projection is split into N *segments*, according to a deterministic segmentation expression over the tuples in a projection (e.g. a hash function over one or more attributes). Segments are distributed over K cluster *nodes*, where each node can contain multiple "local" segments (i.e. $N \geq K$). The resulting layout of a single node is depicted in Figure 6.13. Like SQL Server, Vertica only has update structures for insertion and deletion, and also implements Modify as a full tuple removal plus a reinsertion of the modified tuple.

Tuple data within each segment of a projection is stored in one or more *ROS containers*, each holding a disjoint subset of tuples from that segment, in a column-oriented format and sorted according to the projections sort order. Figure 6.14 shows the low-level storage layout of the earlier segment K, after three inserts and two deletions have been moved into it from WOS by the tuple mover. Clearly, such Moveout operations by the tuple mover result in fragmentation of the storage layout (as does further *partitioning* in a segment, see [LFV+12] for details). Fragmentation leads to more files, more disk seeks, and maybe most importantly, more *merging* during a scan.

To reconstruct a sorted tuple stream during a table scan, a multi-way merge needs to be performed between all ROS containers in a segment. Such value-based merging will quickly become complex and expensive. Therefore, a second responsibility of the tuple mover is the *Mergeout*, which merges several ROS

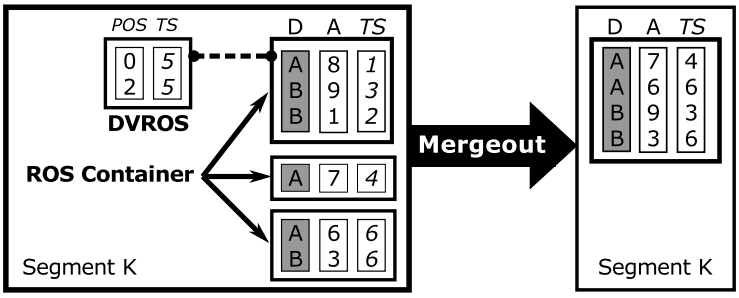


Figure 6.14: Low-level storage layout and Mergeout. Each tuple within a projection is stored in a *ROS container*, which contains one file per column. Within each ROS container, tuples are sorted according to the ordering of the projection. Tuples are implicitly extended with a TS attribute, which adds an extra column that contains the commit timestamp of each tuple. The DVROS is a delete vector that was moved from WOS to ROS, and contains positions of deleted tuples within a ROS container. ROS containers may become fragmented due to Moveout of WOS inserts, or additional *partitioning* within a segment (see [LFV+12]). A *Mergeout* performed by the tuple mover merges ROS containers and compacts out deleted ghost records. In this example, it means reducing the number of files in the segment from 11 to 3.

containers into a large, combined one (the figure shows a rewrite of all ROS containers into a single one, but partial rebuilds may occur in Vertica as well). The Mergeout also compacts out deleted ghost records.

Figure 6.14 also shows a special *timestamp* (TS) column, both for tuples and deletion markers, holding the (logical) commit time of newly inserted or deleted tuples. Vertica uses this cluster-wide global timestamp, in combination with locking, to implement transaction management, using a quorum based commit protocol rather than two-phase commit. Replication of segments over nodes is used for recovery, rather than a transaction log, with rollback of failed transactions relying on removal of WOS entries based on timestamp. The default isolation level is READ COMMITTED, with SERIALIZABLE available as an, obviously more locking intensive, alternative.

Interpreting [LFV+12] literally, table scans in Vertica do not produce the most recently committed table image, as the `Scan` operator only reads (and merges) ROS containers, i.e. only data that was already moved from WOS into ROS by the tuple movers Moveout operation. This means there is no value-based merging of WOS updates during a scan. However, ROS containers still need to be merged using a value-based, multi-way merge, as long as the tuple movers Mergeout does not rebuild the entire table.

Comparing Vertica to Vectorwise, its differentiating advantages are its focus on distributed execution and support for multiple projections. Those may provide a scalable architecture that can be optimized for any well-defined analytic workload. The required update architecture, however, is rather heavy weight.

In terms of raw, non-clustered processing performance we expect Vectorwise to outperform Vertica for multiple reasons. Especially scans are expected to perform better in Vectorwise, with positional merging outperforming the multi-way, value-based ROS container merging of Vertica. Besides the scanning and

processing of sort key columns this requires, Vertica also requires the (compressed) 64-bit timestamp column to be always scanned along and processed. In Vectorwise we only scan the columns of interest to a query, and merge in PDT updates by position. The layering of PDTs always provides an up-to-date and fully isolated table image, while Vertica only provides READ COMMITTED over outdated data (i.e. ROS data only).

The update architecture of Vectorwise is much more light-weight than that of Vertica, as there is no need to deal with multiple projections, distribution, replicas and timestamping. In Vertica, we again see the same inefficiency around Modify updates, implemented as a delete plus insert, discussed in the context of SQL Server in Section 6.9.1. Due to multiple projections, this problem is even amplified. In general, the fact that update operations need to update multiple distinct (and potentially distributed) structures, i.e. separate insert and delete structures, on a per-projection per-segment basis, can be expected to be expensive.

Multiple projections also add significantly to the memory and I/O pressure caused by updates. Although no exact details on WOS data structures and implementation are given, maintaining an in-memory WOS per projection will certainly add to the memory overhead. Also, the tuple movers Moveout and Mergeout operations need to write out tuple data on a per-projection basis. This means that tuple attributes can be expected to be (re)written to disk multiple times before one is back at an "optimal", i.e. unfragmented, compressed table layout (at least twice in case of only a single super projection, i.e. first during Moveout, and eventually by one or more Mergeouts to rebuild a fragmented table). Vectorwise does not suffer from additional I/O due to projections, as projections are not supported. The multi-stage write-out of tuples, however, i.e. first to a "scratch space" to free up WOS memory without immediately undertaking a table rebuild (i.e. checkpoint), could be a good strategy, depending on the workload. There are plans to investigate comparable techniques in Vectorwise in the context of positional updates and SSD storage.

6.9.3 SAP HANA

SAP HANA [FML+12] is a column-oriented data management platform that aims to deliver high performance on both analytic and transactional workloads. It is limited to workloads that reside in main-memory, but can scale out through distributed execution on a cluster [LKF+13]. HANA supports multiple query processing engines to handle data of different degrees of structure (from relational to unstructured graphs and text), thereby supporting multiple domain-specific languages. Query plans in HANA are data-flow graphs that may contain nodes with multiple inputs and outputs, that, besides relational, implement seemingly arbitrary functionality, such as OLAP, graph, text processing, or even custom compiled or scripted operators. Data between nodes may flow tuple-at-a-time, vector-at-a-time or even column-at-a-time (i.e. full materialization), either compressed or uncompressed, depending on what is most appropriate.

All this functionality is implemented on top of a memory-resident *unified table* structure [SFL+12], which provides common table access methods, as illustrated in Figure 6.15. Tuples in a unified table go through a certain "lifecycle", where they enter the system as an insert into the write-optimized *L1-delta*, then propagate through the intermediary *L2-delta*, to eventually end up in the heavily

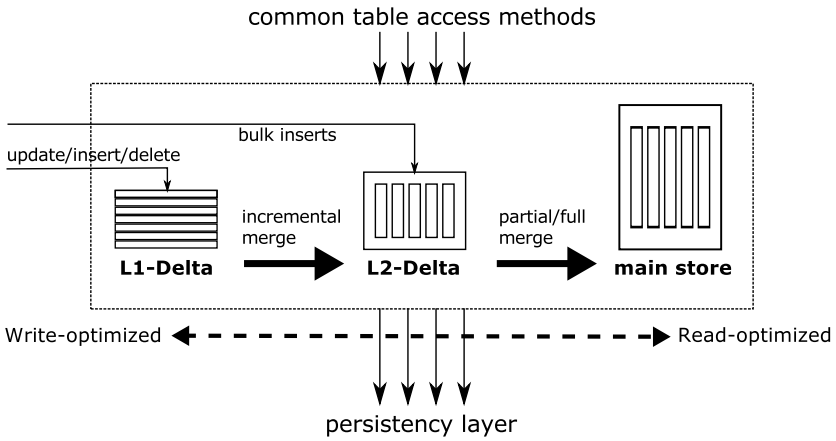


Figure 6.15: Unified Table concept in SAP HANA.

read-optimized *main store*, which contains the bulk of the data.

The L1-delta holds 10.000-100.000 tuples, is row-oriented, uncompressed and allows in-place field updates (i.e. modify). No details are provided about the data structures used to organize L1-deltas, nor about how deletions are represented and organized. Most likely, inserts are simply appended, as HANA tables do not support sorted table storage, while deletions may be organized by some tuple ID.

L2-delta aims to store up to 10 million rows, is column-oriented and lightly compressed using *unordered* dictionary encoding, where each column holds offsets into its dictionary rather than actual attribute values. The advantage of this layout is that new dictionary and column values can simply be appended. This allows bulk inserts to easily bypass L1-delta, going directly to L2. Modify within L2 and main is only supported as a deletion followed by insertion of the modified tuple.

The majority of tuples resides in the heavily scan optimized main store, which uses a column-oriented, heavily compressed table representation. Each column stores bit-packed offsets into an *ordered* dictionary, using the minimal bit-width that can represent the size of the dictionary. The ordered dictionary itself can also be compressed, using a variety of methods. Adding new values to this dictionary is costly, making the main store highly update-unfriendly.

Records propagate to lower layers asynchronously, without interfering with running transactions. During a merge from L1-delta to L2-delta, tuples are decomposed from row format to columnar format, and *appended* to the structures in L2. For every table attribute, values in L1 are looked up in the L2 dictionary for given column, with new values being appended to that dictionary. The corresponding dictionary encoding, i.e. the offset within the dictionary, is appended to the L2 column itself. When this is done, tuples from L1 are truncated, and newly appended L2 data is made visible in a single, atomic operation, taking care to keep old versions around for running operations if needed. Overall, the L1-to-L2-delta merge is simple, cheap and incremental (i.e. append-only).

The L2-to-main-delta merge is resource intensive, as, in general, the entire table layout needs to be rebuilt. I.e. the ordered dictionary of each column has

to be recreated to accommodate new values, which invalidates the existing dictionary offsets in the columns themselves. During this merge, dictionary values that have become obsolete, due to deletions, are discarded. This merge can also be memory intensive, as old copies of both dictionary and column data may have to be kept around in support of concurrent transactions. Optimizations to this "classic merge" are presented in [SFL+12].

Although HANA is a main-memory DBMS, and therefore lacks traditional storage and buffer management, it does provide transaction support with ACID guarantees, which requires some form of support for data persistence. SAP HANA uses multiversion concurrency control (MVCC) to implement isolation. Every update against the L1-delta is logged to a REDO log, while dirty pages in the L2 or main store are periodically flushed to disk using a *savepoint*, which is also marked in the REDO log.

Comparing HANA to Vectorwise, there are some high-level architectural similarities, like the layered approach to dealing with updates, with a goal to amortize the costs to update the larger lower layers. However, a lack of buffer management and disk resident (sorted) table storage, combined with lacking architectural details and comparative benchmark results, make it impossible to perform a meaningful comparison. As with SQL Server, the lack of sorted table storage makes insertion (i.e. append) into L1- and L2-delta fast, and avoids the need for value-based merging during a scan. It is, however, unclear how deletes are handled, and most probably there is still some value-based diff logic involved there. Like Vectorwise's checkpoint operation, HANA's merges seem to support similar concurrency mechanisms, although the recreation of main store tables, in combination with MVCC may introduce large peaks in memory consumption.

In general, the multiplicity of publications around HANA do not shy away from bold claims, positioning it as a "Jack of all trades". To be truly convincing, it would be helpful to see more thorough scientific evidence in support of those claims.

6.10 Summary

We have introduced the Positional Delta Tree (PDT), a new differential structure that keeps track of updates to columnar, compressed (read-optimized) data storage. Rather than organizing the modifications based on a table key (i.e., value- based), the PDT keeps differences organized by *position*. The PDT requires reading less columns from disk than previous value-based differential methods and is also computationally more efficient. We have described algorithms for ACID transaction management using three PDT layers (Read,Write,Trans); allowing efficient lock-free query execution. PDTs can thus be seen as a new and attractive approach to transaction management in column stores, that does not compromise its high analytic read performance.

In the next chapter, we will see how to keep indices up-to-date. Most notably, we provide a solution for efficient maintenance of join indices, which rely heavily on (volatile) positions. An interesting angle for future work is to investigate suitability of the PDT data structure for flash memory (i.e. SSD), to reduce memory pressure from the read-PDT under high update loads.

Chapter 7

Index Maintenance

7.1 Introduction

In the previous chapter, we saw how PDTs can be used to efficiently organize updates against a read-optimized table layout. However, we did not touch the subject of index maintenance. In Section 4.4.4, we described the most important indices in Vectorwise, the *min-max index* and the *join index*, and their reliance on static tuple positions, or SIDs, as logical pointers that allow us to reconstruct (partial) tuples from aligned attribute columns. The reason for using SIDs rather than "current" tuple positions, or RIDs, is that RIDs are constantly changing, and are therefore highly ineffective for use as a referencing mechanism in an index. However, we can not simply restrict indices to static, disk-resident data only. To guarantee that index lookups correctly return all relevant results, even under updates, we also need to incorporate relevant tuple data from differential structures, like the PDT. The main question of this chapter is therefore how to reconcile positional indexing with positional PDT updates. We require index lookups to retain *correctness*, i.e. no false negatives, but are willing to sacrifice some precision, i.e. allowing false positives, if that simplifies index maintenance or lookup.

7.1.1 Outline

We start with a reference of the terminology used in this chapter in Section 7.2. Then we discuss maintenance and lookup mechanisms for both the min-max index and the join index, in Sections 7.3 and 7.4 respectively. In Section 7.5 we provide an outline of the final Insert, Delete and Modify operators. Concurrency issues around index maintenance and concurrent checkpointing operations are discussed in Section 7.6. We conclude the chapter with TPC-H stress-test experiments in Section 7.7 and a summary in Section 7.8.

7.2 Abbreviations and Terminology

In this chapter, we use the following abbreviated notation to describe specifics about PDTs:

(P|C)(R|W|T)PDT$_{t_1-t_9}$

With (P|C) indicating either a parent (P) or child (C) side PDT (i.e. the referenced or referencing side, as implied by a foreign-key relationship between two tables), (R|W|T) indicating a read-, write- or trans-PDT respectively, and finally an optional number indicating a commit timestamp (sequence number). For example, PTPDT$_{t_1}$ represents a parent-side trans-PDT that committed at time t_1. A child-side trans-PDT in the transaction committing directly after that would be named CTPDT$_{t_2}$, which would update CWPDT$_{t_1}$ to CWPDT$_{t_2}$.

Furthermore, we provide the following list of abbreviations for ease of reference. These concepts are explained in more detail as they are introduced in the text.

JI a *join index*, which provides a positional mapping between pairs of tuples from a referenced and a referencing table with matching foreign key attributes.

JIS a *join index summary* index, which provides access into a compressed join index representation by defining so called *sync points*.

sync point an $(FSID, LSID)$ pair in the JIS, representing the SID of a foreign parent tuple (FSID) and the local SID (LSID) of the very first child tuple referring to that parent tuple. A sync-point provides a point of entry to start decoding a compressed join index.

bucket/group the index of the JIS partition that a given parent or child SID falls in. JIS partitions are defined by a sequence of sync points.

+JI a special column in the PDT of a parent (referenced) table, containing a positive increment to the counts in a parent-side JI column.

-JI a special column in the PDT of a parent (referenced) table, containing a negative increment to the counts in a parent-side JI column.

MJI A special *modify join index* update type, used in PDT leaves to mark an incremental update to a count in a JI column, where corresponding $+JI$ and $-JI$ values from value space should be added to the count.

LSID or *local-SID*, represents the SID of a tuple in a *child* table. It is often used together with FSID, and meant to emphasize the distinction.

FSID or *foreign-SID*, represents the SID of the *parent* tuple being referenced by a given child tuple. Note that unlike FRID, an FSID is not guaranteed to map to a unique parent tuple.

LRID or *local-RID*, represents the RID of a tuple in a *child* table.

FRID or *foreign-RID*, represents the RID of the *parent* tuple being referenced by a given child tuple. Contrary to FSID, FRID always refers to a unique tuple in the parent table, as we can not refer to invisible ghost tuples.

SDB$_{\mathbf{cur}}$ the current stable database image, i.e. the disk resident data in the current ‘master‘ image.

$\mathbf{SDB_{new}}$ the new stable database image, as generated by a background checkpoint. When the checkpoint finishes successfully, this becomes the new SDB_{cur}.

$\mathbf{T_{SK}}$ the list of attributes defining the sort key of table T.

7.3 MinMax Maintenance

The min-max index horizontally partitions a table into SID ranges of equal size (see Section 4.4.4), and maintains minimum and maximum values for each attribute within each range.

7.3.1 Update Handling

To keep min-max information accurate, it needs to be maintained under updates. In Vectorwise we choose to only *widen* min-max ranges during insert and modify updates, and to ignore narrowing of ranges during delete. During insert and modify, we have the relevant attribute values at hand, but we have to locate the relevant SID range within the min-max index. After adding an update entry to the trans-PDT, based on RID, we can convert that RID into a storage-level SID of the updated table using Algorithm 11. This algorithm successively calls Algorithm 10 to perform rid-to-sid translation through all three layers of PDTs, using the SID returned by each call as a RID input to the next lower-level PDT, until we end up with a storage-level SID. This final SID can be used to find the containing range within the min-max index, and widen minimum or maximum values for that range if needed.

Algorithm 10 PDT.RidToSid(rid)
PDT.RidToSidLo(rid)

Finds the SID associated with the currently visible tuple identified by given RID. Note that this mapping is always unique, as long as we ignore ghost tuples. The relevant tuple is always at the end of a RID-conflicting delete chain, if that exists.
Version that finds the lowest SID (i.e. the SID of the first ghost tuple under given RID) is omitted. It uses FindLeftLeafByRid and returns the SID of the leftmost leaf entry with matching RID.

1: $(leaf, \delta) \leftarrow this$.FindRightLeafByRid($rid$)
2: $(pos, \delta) \leftarrow this$.SearchLeafForRid($leaf, rid, \delta$)
3: **while** $leaf.sid[pos] + \delta \equiv rid$ and $leaf.type[pos] \equiv$ DEL **do** {Skip delete-chain}
4: $\quad pos \leftarrow pos + 1$
5: $\quad \delta \leftarrow \delta - 1$
6: **end while**
7: **return** $rid - \delta$

This simple approach to min-max maintenance avoids manipulation of tuple positions in the index, as we retain the static partitioning based on storage SIDs. During min-max maintenance, we only manipulate the in-memory min-max structures. When pushing down a selection, the updated min-max will include all SID ranges that contain relevant data, irrespective whether that comes from the PDT or from stable storage. MergeScan then makes sure to merge in all relevant tuples within that range. There are some downsides to this approach, which we discuss next.

Algorithm 11 Table.RidToSid(*rid*)
Table.RidToSidLo(*rid*)
Finds the highest storage SID for a given RID.
Version that finds the lowest storage SID uses RidToSidLo and is omitted.

```
1: sid ← this.tpdt.RidToSid(rid)
2: sid ← this.wpdt.RidToSid(sid)
3: sid ← this.rpdt.RidToSid(sid)
4: return sid
```

First of all, the SID based partitioning into tuple ranges is fixed, meaning that every insert increases the number of tuples covered by the target range. This can result in skew in the index, which, in general, results in more tuples entering the operator pipeline when scanning ranges with many inserts. The novel tuples are scanned from memory though, so no additional I/O is introduced with respect to the situation where we already scan the range.

However, both inserts and modifies may potentially introduce additional I/O during scans, as the min-max index is not able to distinguish between attribute values that reside on disk and those from a PDT. This can only happen after a min-max range actually got widened, which is relatively rare. In case a selection predicate could be satisfied accessing only the in-memory deltas, specifically those responsible for widening the min-max value range, the entire SID range of such min-max entry will still be added to the scan, so that all storage tuples in that range are scanned along, even if none of them satisfies the predicate. For example, if we insert a tuple that increases the max value for attribute x in min-max range R from 'm' to 'n', we end up scanning the full SID range R, even for queries with a selection predicate like $x \geq$ 'n'.

We do not narrow min-max value or SID ranges under deletion, which might lead to false positive tuple ranges during selection push-down. Adding such functionality would severely hamper the Delete operator, as it calls for all columns to be scanned, rather than only those in the delete predicate. For every such column, we would then need to analyze the entire tuple range, to check whether either the minimum or maximum value was impacted. Note that, given the large number of tuples in a (logical) min-max range, it is not likely that minor narrowing of such a range would result in actual I/O benefits, as range boundaries typically do not align with page boundaries on disk. Only long sequences of consecutive deleted tuples, that cover one or more SID ranges in min-max, could result in needless I/Os, as selection push-down will still add those ranges to a scan, even if their tuples later turn out to be deleted by a PDT. This is something that is optimized during a MergeScan though, as it skips over ranges of deleted tuples, avoiding I/Os for pages that are skipped entirely.

Finally, we deal with transactionality of min-max updates in an optimistic way. All transactions simply update the global min-max structure in real-time, protected by a mutex. If a transaction aborts, we can not roll back its changes. This is another source of potential false positives that we accept.

7.3.2 Example

To illustrate min-max maintenance, consider the example `accounts` table and associated min-max from Figure 4.11. Figure 7.1 shows the effect of executing the following mix of updates:

Accounts			
SID	**acctno**	**name**	**balance**
00	019	Isabella	269.38
01	038	Jackson	914.11
02	072	Lucas	**146.61**
03	153	Sophia	266.55
04	156	Mason	850.90
05	282	Ethan	521.60
06	302	Emily	647.38
07	314	Lily	119.40
08	**200**	**Jason**	**449.12**
08	332	Chloe	526.08
09	389	Emma	497.19
10	533	Aiden	22.03
11	592	Ava	140.67
12	808	Mia	383.69
13	896	Jacob	899.41

Accounts.MinMax							
bucket	**SID**	**acctno**		**name**		**balance**	
		min	**max**	**min**	**max**	**min**	**max**
0	00	019	153	Isabella	Sophia	**146.61**	914.11
1	04	156	314	Emily	Mason	119.40	850.90
2	08	**2**00	592	Aiden	**Jason**	22.03	526.08
3	12	808	896	Mia	Jacob	383.69	899.41

Figure 7.1: MinMax index for `accounts` table after updates

```
UPDATE accounts SET balance = balance - 200 WHERE acctno = 72
DELETE FROM accounts where acctno > 155 AND acctno < 325
INSERT INTO accounts VALUES (200, Jason, 449.12)
```

Modified or new values are shown in boldface, deleted tuples are grayed out. We see that the SQL UPDATE changed the min value in the lowest min-max tuple range (i.e. bucket 0). The DELETE removed all tuples in bucket 1, but did not change anything in min-max. The final INSERT is interesting, in that it is an "out-of-order" insert, ending up right before the tuple with `acctno` 332, where it belongs, but *after* the chain of deletes. Based on the ordering on `acctno`, the tuple should belong to bucket 1. But because of the chain of deletes, it now ends up in bucket 2, changing the min value for `acctno` in that bucket. Over time, even min-max buckets of a sort attribute might start to overlap.

Rerunning the earlier `SELECT * FROM accounts WHERE name > 'Lara' AND balance < 200` from Section 4.4.4, the min-max now tells us that buckets 0, 1 and 2 all qualify, so we end up scanning the range [0, 12), which boils down to scanning all 12 I/O blocks from Figure 4.10.

This example shows how min-max can get polluted over time, and what the impact on query processing can be. Such negative effects are eliminated by a checkpoint, when the disk resident image of a table, and its min-max index, are rebuilt. In reality, with larger tables and millions of tuples per bucket, things do not deteriorate as rapidly as in this small example. For example, during TPC-H update stress tests, discussed later in Section 7.7, we did not measure any additional I/O caused by min-max pollution.

7.3.3 Range Scans

Min-max can be used to restrict scans to relevant ranges of a table. However, it only gives us SID ranges. To map these SID ranges to (SID, RID) ranges, which are required to unambiguously initiate scan positions within a PDT, we still need mechanisms to convert SIDs to RIDs. SIDs are not guaranteed to be unique, as PDT inserts share the SID with the tuple before which they got inserted. As with rid-to-sid conversion, we therefore create "low" and "high" variants of sid-to-rid conversion, that return the first and last RID associated with a SID chain, respectively, as shown in Algorithm 12. Repeated application of this algorithm through the read-, write- and trans-PDT can then be used to convert storage level SIDs to top level RIDs, as outlined in Algorithm 13.

Algorithm 12 PDT.SidToRid(sid)
PDT.SidToRidLo(sid)
Finds the RID belonging to the original tuple (non-PDT-insert) with given SID. This is always the tuple at the end of a potential SID-conflicting insert chain, and could actually be a ghost, in case the original tuple is deleted.
Version that finds the lowest RID (i.e. the RID of the first PDT insert under given SID, if any) is omitted. It uses FindLeftLeafBySid and returns the RID of the leftmost leaf entry with matching SID.

1: $(leaf, \delta) \leftarrow this$.FindRightLeafBySid($sid$)
2: $(pos, \delta) \leftarrow this$.SearchLeafForSid($leaf, sid, \delta$)
3: **while** $leaf.sid[pos] \equiv sid$ and $leaf.type[pos] \equiv$ INS **do** {Skip insert-chain}
4: $pos \leftarrow pos + 1$
5: $\delta \leftarrow \delta + 1$
6: **end while**
7: **return** $sid + \delta$

Algorithm 13 Table.SidToRid(sid)
Table.SidToRidLo(sid)
Finds the highest RID for a given storage SID.
Version that finds the lowest RID uses SidToRidLo and is omitted.

1: $rid \leftarrow this.rpdt$.SidToRid(sid)
2: $rid \leftarrow this.wpdt$.SidToRid(rid)
3: $rid \leftarrow this.tpdt$.SidToRid(rid)
4: **return** rid

To guarantee correctness of min-max range lookups, we have to make sure that all relevant tuples are present in a range R. That is, both the stable tuples from R, together with all PDT resident updates applicable to R. We therefore convert SID-range R to its widest possible corresponding RID range, to avoid false negatives, by using SidToRidLo to convert the low end of R, and SidToRid on the high end.

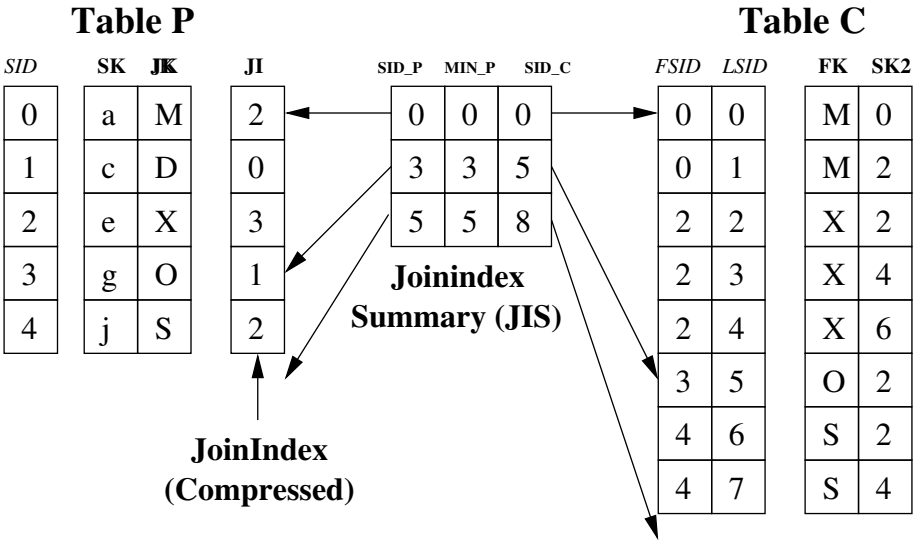


Figure 7.2: Compressed join index connecting a parent table (P) and a child table (C)

7.4 Join Index Maintenance

7.4.1 An Updateable Join Index Representation

The clustered, positional join index (JI) employed by Vectorwise was described in Section 4.4.4, and depicted in Figure 4.12. The layout proposed there is inconvenient and inefficient to maintain under updates to either of the related tables. For example, inserting or deleting a tuple in the ordered sequence of either the child or the parent table would require half the join index entries to be shifted, on average, to accommodate for an inserted or deleted JI entry, and invalidates all positional references below that entry.

Figure 7.2 represents a more compact and update-friendly representation of a join index, where it is reduced to a single column, labeled `JI`, with an arity equal to that of the parent table, P. Recall that P has two columns, with SK being used as the sort key, and JK being used as the "join key", which is referenced by the foreign key attribute FK in child table C. The child is organized according to the sort order of the parent, i.e. *clustered*, through the FK relationship. It has a second attribute, SK2, which acts as a secondary sort attribute. Overall, it is therefore sorted on (P.SK, C.SK2) [1]. The SID, FSID and LSID columns are virtual columns that are not stored. Treatment of the *join index summary* JIS structure is deferred to Section 7.4.3.

The virtual FSID, or *foreign SID* column contains for each child tuple the SID of the matching parent tuple. We use LSID, or *local SID* to refer to regular child-side SIDs, emphasizing the difference from FSID. The JI column is simply a *run length encoded* (RLE) representation of FSID, where we store at offset (or

[1] The fact that a child table is sorted on parent attribute(s) is the main reason we decided to drop storage of SK values for ghost tuples from our original design [HZN+10], rather accepting the fact that inserts might be out-of-order with respect to deleted tuples (see Section 6.4.7).

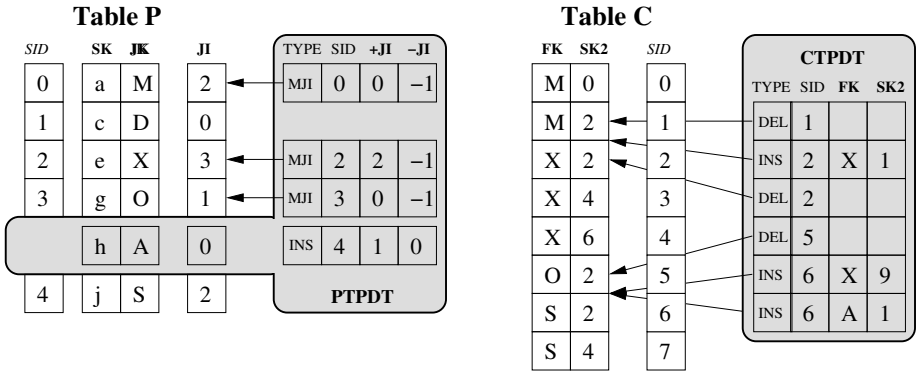


Figure 7.3: Example of join index maintenance caused by updates to both the parent (PTPDT) and child (CTPDT) side. For simplicity, we consider the read- and write-PDT to be empty, but the idea extends to stacked PDTs.

SID) i, the number of occurrences of i in FSID. I.e. JI encodes 2 instances of 0, 0 instances of 1, 3 instances of 2, 1 instance of 3, and 2 instances of 4, which matches exactly the sequence in FSID. One can also interpret JI to contain, for each parent tuple, the number of referencing child tuples, i.e. a *reference count*.

This correspondence is crucial to understanding our approach to join index maintenance, as child side inserts or deletes boil down to proper adjustments of these reference counts. If we treat the JI column as a special column of table P, insertion of a new tuple into the PDT of P sets the initial JI count for that tuple to 0, while during deletion we can use the count to enforce referential integrity, requiring the count of a deleted parent tuple to be 0.

7.4.2 Join Index Updates

Now that we added the join index (JI) as a special column to parent table P [2], we can handle join index updates through the PDTs of P. The initial JI column is filled when child C is bulk-loaded with stable data. We extend the PDT of P with a new type of update, a *modify join index* (MJI). Such an MJI update is similar to a regular modify, except it does not *replace* the original value, but rather *increments* an integer column with a given amount, as specified by the MJI update value. This way, we can alter the counts in the JI column by manipulating increments in the parents trans-PDT.

Figure 7.3 shows an example of updates involving the join index. In the (simplified) PDT of the parent, PTPDT, we see three updates of type MJI, and the SID they apply to. With each MJI update, we associate *two* value columns (in value space), +JI and -JI, containing a positive and a negative *increment* to the join index, respectively. The reason we split increments into a positive and a negative component will be given in Section 7.6.3. For now, it suffices to know that the +JI of parent tuple τ is incremented by one for each inserted child tuple referencing τ, while -JI is decremented by one for each deleted child tuple referencing τ. The current JI counts can then be obtained by adding both

[2]It is possible to have multiple JI columns in a parent table, one per incoming join index. A child table can only have a single outgoing (clustered) join index, as it enforces an ordering of the child.

+JI and -JI to the stable JI entries from disk. We extend our Merge operator (Algorithm 3) to handle updates of type MJI in such an incremental way.

Looking at the child PDT, CTPDT, in Figure 7.3, we see a DEL of SID=1, which refers to JK='M' in the parent. Therefore, the tuple at SID=0 in the parent has its -JI field decremented by 1. Similarly, the INS at SID=2, contributes 1 to the increment of +JI for the parent tuple with JK='X'. After this, we have two deletes, involving the foreign tuples with JK 'X' and 'O', respectively, decrementing their -JI by 1. The following insert is interesting, as it is an "out-of-order" insert, with respect to the ordering of the earlier deleted tuples. It refers back to the parent tuple with JK='X', which we already touched during the previous insert, completing its +JI of 2 that we observe. The final insert refers to a new parent tuple that was inserted in the same transaction, with JK='A'. This parent tuple defaults to an initial JI count of zero, which can be altered through the +JI and -JI increments.

The up-to-date join index produces current *foreign-RID* (FRID) and *local-RID* (LRID) pairs, rather than (FSID, LSID). To reconstruct it, we extend our Merge algorithm from Algorithm 3 to support incremental MJI updates. When scanning P.JI, MergeScan first uses Merge against all three layers of PDTs, providing an up-to-date version of the *compressed* join index, which stores at offset (or parent RID) i the number of occurrences C of i in the child's virtual FRID column. In a second phase, MergeScan can decompress the counts into C.FRID, by outputting C instances of i. For our example in Figure 7.3, the up-to-date JI counts would be 1, 0, 4, 0, 1, 2. Decompressing it gives an FRID sequence of 0, 2, 2, 2, 2, 4, 5, 5. The corresponding child-side LRIDs are simply the enumeration of this sequence, 0, 1, ..., 7.

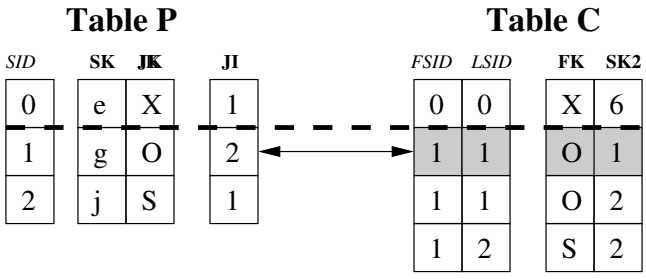
7.4.3 Join Index Summary (JIS)

To find the foreign-RID (FRID) belonging to an arbitrary child-side tuple at LRID=i, we need to sequentially scan and decompress the parent-side JI column from the start, until we reach the i'th decompressed value (counting from 0), or, alternatively, sum the JI counts until we find the bucket that covers i. For range scans, this is undesirable. The *join index summary* (JIS), already depicted in Figure 7.2, alleviates this problem by providing a clustered, sparse index on (FSID, LSID), i.e. the uncompressed image of the the stable part of the JI column, thereby providing *sync points* for the decompression process. JIS partitions the SID space of table C into SID ranges, in such a way that the start of a partition coincides with a change of value in C's foreign key column, C.FK, which we call a *cluster head*.

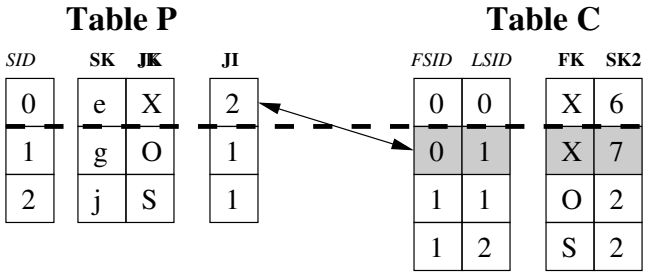
cluster head refers to the first child tuple, in terms of sort key ordering, within a cluster of child tuples that match on foreign key.

sync point is a special (FSID, LSID) pair that marks the start of a JIS partition, with the requirement that LSID refers to a cluster head. A consequence is that parent tuples that are not referenced by any stable child tuple, and therefore have a stale P.JI count of 0, are never used as a sync point.

For each partition, JIS stores LSID, as SID_C, and the corresponding FSID, as SID_P, which can be used to match the count in P.JI to its first child-side reference, allowing us to initiate decompression of the join index.



(a) Safe insert



(b) Dangerous insert

Figure 7.4: PDT inserts, shown in gray, that share the SID of a sync point (dashed line). Note that the join index column (JI), shows *up-to-date* values.

Unfortunately, we can not always rely on a sync point to provide a valid offset to start a scan. The reason being that JIS indexes SIDs, and SIDs are not guaranteed to be unique as soon as PDT inserts are involved This might lead to FSID references crossing the boundary defined by a sync point, rendering that sync point invalid. In Figure 7.4 we see a minimal example of such a scenario.

Figure 7.4 shows two distinct inserts into the child table, both ending up at the same child position, at LSID=1. We do not show the entire JIS structure, but restrict ourselves to a single partition boundary, assumed to be defined in the JIS, but here only marked by the dashed line. This corresponds to a sync point of (FSID=1, LSID=1), meaning that both child inserts share their LSID with the sync point.

The crucial difference between the inserts (shown in gray) in Figure 7.4a and 7.4b is that they refer to different tuples in the parent. Both ('O', 1) and ('X', 7) sort directly before the ('O', 2) sync point. However, the ('O', 1) tuple belongs to the *same* FK cluster as the stable ('O', 2) tuple, in fact introducing a new cluster head, one that shares its SID with the former ('O', 2) cluster head. Given that the ('O', 1) insert also increments the JI count of the parent-side sync point, at FSID=1, this sync point remains valid, or *stable*.

The ('X', 7) insert, on the contrary, is the new tail of the 'X' cluster that originates in the *preceding* partition, as indicated by its reference to FSID=0. We can not avoid this tuple from acquiring an LSID of 1, by definition of the way we handle inserts, and have to accept that updates can cause an FK-cluster to overlap a sync point. This scenario renders a sync point invalid, or *dirty*. A dirty sync point can not be used to initiate decompression, as the ('X', 7) tuple would contribute towards the JI count of the parent-side sync point, at FSID=1,

which would corrupt the decompression.

For completeness, note that one scenario is lacking from Figure 7.4: a scenario where a child-side tuple, γ, in some partition P_i, refers to a parent tuple, π, in a *succeeding* partition, P_{i+1}. Assuming that insertion of child-side tuples respects the joint sort-key ordering, such a situation can only arise in case the parent-side sync point of P_{i+1}, say tuple σ, is marked deleted, with π being the first visible parent tuple directly after that deletion. Now assume that in such a scenario we could insert tuple γ, which refers to π, into P_i. This would require all tuples referring to σ to still be present in the child, to force γ into P_i in terms of sort ordering. However, due to referential integrity, such a scenario can not occur, as all tuples referring to σ need to be deleted before we can delete σ. This forces γ to the end of the corresponding delete chain in the child, moving γ into π's partition, P_{i+1}. (Recall that parent tuples that are not referenced, i.e. with a stable JI count of 0, are ignored in the construction of sync points in the JIS).

JIS Maintenance and Lookup

Given that PDT updates can render a sync point invalid, we need a JIS maintenance mechanism that marks dirty sync points, together with lookup mechanisms that avoid usage of them. This is why we have a third field, MIN_P, in the JIS layout of Figure 7.2. For each JIS partition, MIN_P holds the *current* minimum parent SID referenced by child tuples in that partition, and is the only field in JIS that can change under updates. SID_P and SID_C are kept stable.

MIN_P starts out with a value equal to SID_P. As soon as $MIN_P < SID_P$, the sync point for given partition is invalid. Note that MIN_P can only become smaller than SID_P, as we learned in Section 7.4.3 that child-side FK references in partition P_j can only refer to partitions P_i such that $i \leq j$. Also note that, in general, i can indeed be smaller than $j - 1$, meaning that child tuples can refer to parent tuples in *any* preceding partition [3]. This happens if a block of consecutive child-side tuples that covers more than one sync point is deleted, after which new child-side inserts that sort within the deleted range end up as PDT inserts at the tail of that delete chain. Such an out-of-order insert renders all the sync points in the deleted range invalid, by updating their MIN_P field to the FSID of the newly inserted tuple.

A JIS index can be maintained by passing it an (FRID, LRID) pair for every child-side insert. Such a pair represents the RID of the parent tuple being referenced, FRID, together with the child-side insert position, LRID. It is generated during a (very efficient) [4] join between the parent and child table, required to find the correct insert position within the clustered child. After converting (FRID, LRID) into (FSID, LSID), using RidToSid (Algorithm 11), we locate the JIS partition LSID falls into, and update MIN_P in case it is bigger than FSID.

When performing a JIS lookup, the goal is to, given either an FSID or an LSID, locate the nearest preceding stable sync point, which allows us to initiate

[3] Which is the reason for maintaining MIN_P SIDs for every sync point, rather than a simple "dirty bit".

[4] This join can be computed by a linear scan of the parent- and child-side sort key attributes, together with a decompressed scan of the current join index.

Algorithm 14 JIS.findSyncPoint($fsid$)
For an arbitrary parent SID, $fsid$, this routine searches backwards through the partitions in a JIS, until a non-dirty sync point is found, and returns the corresponding stable sync point, (SID_P, SID_C).

```
1: sidP ← 0
2: sidC ← 0
3: for i ← this.size() − 1; i >= 0; i ← i − 1 do
4:    sidP ← partition[i].SID_P
5:    sidC ← partition[i].SID_C
6:    if partition[i].MIN_P ≤ fsid then
7:       break {Note that MIN_P is unique due to FK clustering}
8:    end if
9: end for
10: while i > 0 and partition[i].SID_P ≠ partition[i].MIN_P do
11:    i ← i − 1
12:    sidP ← partition[i].SID_P
13:    sidC ← partition[i].SID_C
14: end while{Search backwards for nearest non-dirty sync-point}
15: return (sidP, sidC)
```

a range scan. For a given FSID, this process is outlined in Algorithm 14. The algorithm first searches the JIS for the partition that covers $fsid$, and returns the nearest non-dirty sync point it finds. In case of a child-side LSID, we first locate the partition it falls into, and use the MIN_P field of that partition as an input to Algorithm 14.

7.4.4 Range Propagation

We now have the basic ingredients to implement range propagation, both from parent to child and from child to parent. Range propagation can either stem from a SID range, as introduced, for example, by selection push-down using a min-max index, or from RID ranges, as introduced by range partitioning on the current table count during parallelization of query plans. The resulting parent and child ranges respect the original clustering, allowing us to employ highly efficient merge-based join plans.

We restrict our discussion to the most complex, and interesting, range propagation scenario: from a child-side RID range to the corresponding parent RID range. Our goal is to generate the virtual FRID column for a given child-side range, i.e. for each child tuple in the range, fill in the RID of the matching tuple in the parent, without performing a join. This requires range propagation to find the corresponding range in the parent, as that is where the JI column and its updates are stored. Furthermore, the start of the parent range gives us a JI count, but we likely have to start decompressing at a certain offset within that count, as the corresponding child tuple is not necessarily the first in its cluster. The process is outlined in Algorithm 15, where we restrict ourselves to the start offset of the RID range, $ridC$, as the end is simply a matter of counting.

Algorithm 15 takes as input a child RID, $ridC$, which it first converts to the SID associated with that tuple, $sidC$. At line 2, we then convert this child

Algorithm 15 JI.initializeDecompressedScan($ridC$)
Initialize a scan over a virtual FRID column, starting from an arbitrary LRID in the child, provided as $ridC$.

1: $sidC \leftarrow child.RidToSid(ridC)$
2: $minP \leftarrow JIS.childLoToParentLo(sidC)$
3: $(sidP_{sync}, sidC_{sync}) \leftarrow JIS.findSyncPoint(minP)$
4: $ridP_{sync} \leftarrow parent.SidToRidLo(sidP_{sync})$
5: $ridC_{sync} \leftarrow child.SidToRidLo(sidC_{sync})$
6: $skipC \leftarrow ridC - ridC_{sync}$
7: $this.initScan(ridP_{sync})$
8: $this.skipScan(skipC)$
9: **return**

SID to a parent SID. For this, we use JIS.childLoToParentLo($sidC$) [5] , which finds the partition $sidC$ falls into, and returns the current value of the MIN_P field in the corresponding JIS entry, which is a guaranteed lower-bound for the parent-side SID we are searching for. We then use Algorithm 14 to convert this (potentially dirty) $minP$ SID to the nearest stable sync point.

Now that we have a stable SID sync-point, we need to convert it to a conservative RID sync point, by including any PDT inserts that reside at $sidP_{sync}$ in the parent or $sidC_{sync}$ in the child. Note that the fact that the sync point is stable, guarantees us that none of the child inserts refers to an earlier partition, so that the first of them is a cluster head, either for the original parent-side sync tuple, or for a newly inserted one (at $sidP_{sync}$).

Given the pessimistic RID sync point, at line 6 we compute how many tuples we can skip to reach our $ridC$ of interest. We then initiate a (decompressing) join index scan from the safe sync RID. The skipScan routine then performs a Merge of the JI column, producing up-to-date counts, discarding the first $skipC$ worth of cumulative counts. Now the join index scan is positioned at the destination $ridC$, and we are ready to produce uncompressed FRIDs.

7.5 Update Operators

Now that we know how to add updates to a PDT and the impact they might have on indexing structures, we are ready to provide a high-level outline of the full update operators, Insert, Delete and Modify.

7.5.1 Insert

The Insert operator adds a batch of tuples to a table. The batch should be sorted according to the sort key ordering of the destination table, and enumerated by the RID positions to insert each tuple at. The RID positions can be obtained by MergeFindInsertRID (see Section 6.4.6), the output of which can be fed directly into Insert. If we insert into the child side of a join index association, each new tuple should furthermore be annotated with the parent-side RID (FRID) of the

[5] The naming of this routine indicates that we are converting the low end of the child-side range, and wish to convert this to a conservative (safe) lower bound in the parent. Similar routines exist for high ends, and also for the inverse direction, from parent to child.

tuple it refers to. These FRIDs are obtained from a foreign-key join between the insert batch and the parent table.

Algorithm 16 Table.Insert(*tuples*, *rids*)
Inserts an ordered batch of *tuples* into a Table (*this*) at the RID positions given in *rids*. The batch should be ordered on the sort key of the destination table, which implies that *rids* is non-decreasing. If *this* acts as the referencing side in a join index relationship, each tuple must be annotated with $FRID$, the parent-side RID of the tuple being referenced.

```
1:  i ← 0
2:  for (tuple, rid) in (tuples, rids) do
3:    rid ← rid + i
4:    tpdt.AddInsert(rid, tuple)
5:    sid ← this.RidToSid(rid)
6:    minmax.updateAll(sid, tuple)
7:    if isJoinParent(this) then
8:      tpdt.InitJoinIndexCounts(rid)
9:    end if
10:   if isJoinChild(this) then
11:     frid ← tuple["FRID"]
12:     ji.parent.tpdt.IncrementJoinCount(frid, 1)
13:     fsid ← ji.parent.RidToSid(frid)
14:     lsid ← this.RidToSid(rid)
15:     ji.jis.TestAndSetMinForeignSid(fsid, lsid)
16:   end if
17:   i ← i + 1
18: end for
```

Algorithm 17 JIS.TestAndSetMinForeignSid($fsid, lsid$)
Checks whether we should update the MIN_P field of the JIS partition $lsid$ falls into. If $fsid$ is smaller than the current MIN_P value, we update it to $fsid$.

```
1: partitionIdx ← lsid/this.partitionSize
2: mutex_lock(this.mutex)
3: if fsid < this.partition[partitionIdx].MIN_P then
4:   this.partition[partitionIdx].MIN_P ← fsid
5: end if
6: mutex_unlock(this.mutex)
```

The Insert operator itself is outlined in Algorithm 16, where we iterate over the tuples in the insert batch. We add each tuple to the PDT, adjusting the insert-RID by i to accommodate for the shift introduced by tuples inserted during earlier iterations. The next step finds the corresponding SID, and uses it to update the global min-max index of the destination table (using a mutex for protection from concurrent modifications).

If the destination table participates as a parent in one or more join index associations, we initialize the join index (JI) counts to zero. For child-side inserts,

we increment the +JI field of the referenced parent tuple, at FRID, by one, to account for the new reference. Finally, we convert both FRID and the local insert RID, *rid*, to (FSID, LSID), which we pass to the JIS.TestAndSetMinForeignSid routine (Algorithm 17) to maintain MIN_P of the JIS partition LSID falls into [6]. As with min-max, the JIS index is maintained "optimistically", meaning that we directly manipulate the global data structure, accepting potential pollution in case a transaction happens to abort.

7.5.2 Delete

Delete is similar to Insert in that we need to manipulate join index counts. However, we do not perform maintenance on min-max and JIS indices. When deleting from the parent table in a join index association, we should ensure that referential integrity constraints are not violated. I.e. a parent tuple may not have any child-side references at the time we try to delete it. Given the reference counts in the JI column, we can easily verify that the current count is 0, as is done for all incoming join indices in Algorithm 18.

Algorithm 18 Table.Delete(*rids*, $frids$)
Deletes the tuples at RID positions given in *rids* from a table (*this*). The optional $frids$ argument must be provided in case we delete from the referencing (i.e. child) side in a join index association, and should contain, for each deleted tuple, the parent-side RID of the tuple being referenced.

```
1:  qpdt = pdt_create()
2:  for (rid, frid) in (rids, frids) do
3:    if isJoinParent(this) then
4:      for jiColumn ← this.NextJoinIndexColumn() do
5:        if jiColumn.GetJoinCount(rid) ≠ 0 then
6:          return "ERROR: referential integrity violation"
7:        end if
8:      end for
9:    end if
10:   if isJoinChild(this) then
11:     ji.parent.tpdt.DecrementJoinCount(frid, 1)
12:   end if
13:   qpdt.AddDeleteBySid(rid)
14: end for
15: tpdt.Propagate(qpdt)
16: qpdt.destroy()
```

When deleting from a child-side table, for every deleted tuple we also need to know the foreign-RID (FRID) that identifies the referenced tuple in the parent. Those FRIDS can be readily obtained by scanning along the up-to-date and decompressed join index in a Delete plan. They are used in the call to DecrementJoinCount to decrement the -JI field of the referenced tuple in the parents trans-PDT.

[6] In a real-world "vectorized" implementation, we first gather a batch of (FSID, LSID) pairs to amortize the locking overhead associated with updating the global JIS structure.

Algorithm 18 also shows the usage of a fourth PDT layer, the *query-PDT*, identified by *qpdt*. It starts out empty, and contains updates with respect to the RID image produced by the current trans-PDT, i.e. the SIDs in *qpdt*, refer to the RID enumeration generated by a merge of the trans-PDT. The purpose of the query-PDT is to provide a query-local isolation layer to effectively sort an arbitrary (i.e. unordered) sequence in *rids* on the fly. Recall that for Insert, where new tuples either come in sort-key order, or are appended to the end of a table, we had to adjust the destination RID to compensate for previously inserted tuples, allowing in-place modification of the trans-PDT. If such an ordering can not be assumed, we avoid direct manipulation of the trans-PDT, and treat input *rids* as SIDs of the query-PDT, as illustrated by our use of AddDeleteBySid. When all input RIDs are processed, we use Propagate to migrate the updates from the query-PDT into the trans-PDT.

7.5.3 Modify

All we need to do in case of Modify, is to add a PDT update for each attribute being altered, and to inform the min-max index about the changes to relevant columns, so that it can check for changes to minimum or maximum attribute values in the relevant SID range. The process is summarized in Algorithm 19. Modify never changes the SID or RID enumeration of tuples, and modifications of sort key attributes are rewritten into Delete followed by Insert.

Algorithm 19 Table.Modify(*colnos*, *valueLists*, *rids*)
Updates a list of attributes identified by *colnos*, for all tuples at positions in *rids* with the corresponding attribute values from *valueLists*.

```
1: for (valueList, rid) in (valueLists, rids) do
2:   for i = 0; i < colnos.size(); i = i + 1 do
3:     tpdt.AddModify(rid, colnos[i], valueList[i])
4:     sid ← this.RidToSid(rid)
5:     minmax.updateColumn(sid, colnos[i], valueList[i])
6:   end for
7: end for
```

7.6 Concurrency Issues

In Section 7.5 we described optimistic maintenance of the global min-max and JIS indices belonging to a table, where we used simple mutual exclusion mechanisms to avoid corruptions caused by concurrent updates. There are, however, more subtle concurrency issues that are semantic in nature, as they are caused by the inherent volatility of positional information under updates. Section 7.6.2 discusses the issue of maintaining indices in a second database image, as generated during a background checkpointing transaction. In Section 7.6.3 we discuss obstacles during serialization of trans-PDTs from the child-side of a join index association. Solutions to both problems rely on a generic solution to the problem of matching child-side PDT inserts to the (volatile!) foreign-RID of the parent tuple they reference, at any moment in time, without performing a join. Therefore, Section 7.6.1 first presents a solution to that problem.

7.6.1 Computing FRID for Child-PDT Inserts

Given a child-side PDT, $\text{CPDT}_{t_b}^{t_a}$, together with a parent-side PDT of the table being referenced, $\text{PPDT}_{t_d}^{t_c}$, we can compute the FRID associated with each CPDT insert at any moment in time, say t_n, as long as we have that $t_a \equiv t_c$ and $t_b \equiv t_d \equiv t_n$, i.e. both PDTs hold updates from exactly the same time interval. As long as this requirement is satisfied, the +JI and -JI counts of a tuple τ in the PPDT give us the exact number of inserts and deletes that refer to τ from CPDT. Combining this with the clustering property of our join indices, i.e. the child being ordered according to the sort key ordering of the parent, we can reconstruct the FRID of each CPDT insert by means of an iterative process, performing a left-to-right traversal through the leaves of both the PPDT and CPDT, in a lock-step fashion. To iterate the parent-side join index updates (MJI), we introduce a JiIncrementsIterator. Once initialized, we iterate through the updates in the leaves of CPDT, and notify the JiIncrementsIterator after we encounter a child-side insert. This allows the JiIncrementsIterator to progress along, providing access to both SID and RID of the matching parent tuple, for every child-side insert we encounter. The logic for this is shown in Algorithm 20.

Algorithm 20 JiIncrementsIterator.nextChildInsert()
Advances this parent PDT iterator after encountering an insert in the corresponding child PDT.

Require: Iterator class variables should have been properly initialized to a state where *pos* has been advanced from 0 until the first parent PDT (*ppdt*) entry with a non-zero +JI field (using a loop similar to the one below).

1: $insCount \leftarrow insCount + 1$
2: **while** $insCount \equiv jiIncrement$ **do** {find next non-zero +JI entry}
3: **if** $ppdt.type[pos] \equiv INS$ **then**
4: $\delta \leftarrow \delta + 1$
5: **else if** $ppdt.type[pos] \equiv DEL$ **then**
6: $\delta \leftarrow \delta - 1$
7: **end if**
8: $pos \leftarrow pos + 1$
9: **if** $pos \equiv ppdt.count()$ **then**
10: **return false**
11: **end if**
12: $insCount \leftarrow 0$
13: $curFSID \leftarrow ppdt.sid[pos]$
14: $curFRID \leftarrow curFSID + \delta$
15: $jiIncrement \leftarrow ppdt.getJoinCountIncrement(curFRID)$
16: **end while**
17: **return true**

Ensure: $insCount \leq jiIncrement$

A call to Algorithm 20 after a CPDT insert increments *insCount*, which is then compared to the total number of referring inserts, $jiIncrement$, for the current parent tuple. Once those referring inserts are accounted for, *insCount* is reset to zero, and the parent-side cursor is advanced to the next tuple with one or more referring child-side inserts, i.e. a non-zero +JI field, updating $jiIncrement$ accordingly, and keeping track of the running delta. The current parent-side SID

and RID are maintained as $curFSID$ and $curFRID$, respectively, allowing the child to retrieve those values whenever needed.

Running this algorithm over the CTPDT from our earlier example in Figure 7.3, we see three inserts, two that refer to the parent tuple with JI='X', and one that refers to JK='A'. Looking solely at the CTPDT information, there is no way of deducing the corresponding foreign-RIDs. However, using the JiIncrementsIterator, we can correlate those inserts with the two non-zero +JI entries in PTPDT. The first two inserts are accounted for by the parent tuple at FRID=FSID=2, due to its +JI increment of 2. The JK='X' matches the child side foreign keys, FK='X'. After those two inserts, the JiIncrementsIterator advances to the parent tuple at FRID=FSID=4; the newly inserted parent tuple with JI='A'. Again, the child tuple has a matching FK. We now have FRID/FSID information for each child-side insert, deduced from memory-resident information only, i.e. there is no need to access disk resident attribute values, as would be required by a foreign-key join.

This example also motivates our choice to split the PDT-resident increments associated with a modify-join-index (MJI) update into a separate +JI and -JI field. If we would rather maintain a single JI_δ field, to be incremented on child-side inserts and decremented on child-side deletes, we loose the necessary detail to efficiently correlate child-side inserts with their parent-side counterpart. For example, in Figure 7.3, if we would sum +JI and -JI, the resulting JI_δ of -1 at parent SID=0 could match either only the first child-side delete, or the initial child-side DEL-INS-DEL sequence. Both add up to -1, and recovering from a bad choice at a later moment would be complex and inefficient.

7.6.2 Deferred Index Maintenance

The first use case of Algorithm 20 is related to background checkpointing transactions. Recall that during a checkpoint (Section 6.6), where we rebuild the current stable database (SDB_{cur}), plus updates from locked down read-PDTs, into a new disk resident database image (SDB_{new}), we allow concurrent update transactions to start or commit. Update transactions that overlap with a checkpoint update the global min-max and JIS indices in SDB_{cur} directly. However, their changes also need to be reflected in SDB_{new}, a requirement that is complicated by the fact that the new indices are based on a changed SID enumeration, and that they might not even have been rebuilt yet at time of update.

The time line in Figure 7.5 summarizes the potential overlap scenarios between a background checkpoint and a concurrently executing transaction. In the first scenario, tx_1 is already running when CP starts, and commits while the checkpoint is still running. Any inserts to a join child table in tx_1 will directly update the corresponding JIS in SDB_{cur}, by passing every matching (FSID, LSID) pair to JIS.TestAndSetMinForeignSid (Algorithm 17). In case another transaction commits between $start(tx_1)$ and $start(CP)$, the SIDs in tx_1 are not valid in SDB_{new}, so we can not generally apply the updates from tx_1 to SDB_{new} when the checkpoint finishes at $commit(CP)$. The second scenario, where tx_2 is fully overlapped by the checkpoint, suffers from a similar problem, except that its trans-PDT updates are, in general, against a database image that is *newer* than what is being checkpointed. This is the case if one or more transactions committed between $start(CP)$ and $start(tx_2)$.

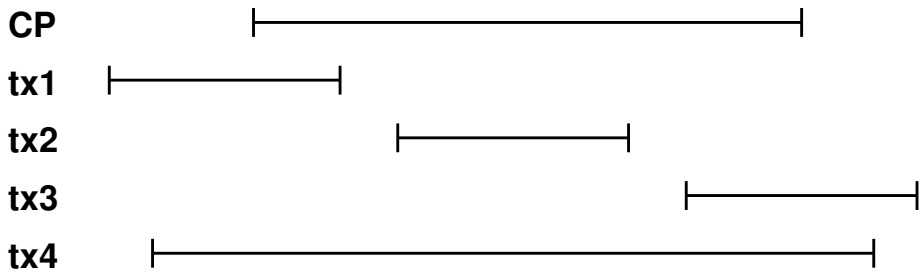


Figure 7.5: Time line showing overlap scenarios of transactions with a concurrent background checkpoint (CP)

For both tx_1 and tx_2, we have to delay maintenance of any JIS indices till checkpoint commit, at $commit(CP)$, and make sure that all (FSID, LSID) pairs are relative to the checkpointed SID image at $commit(CP)$. Recall that all updates that commit during a checkpoint, are gathered in the write-PDTs of respective tables, and that propagation of write-PDT updates into the read-PDT is blocked as long as a checkpoint is running (Figure 6.7). This guarantees us that, on $commit(CP)$, for every checkpointed table, we have a write-PDT that holds all updates committed between $start(CP)$ and $commit(CP)$. Furthermore, the SIDs in those write-PDTs are compatible with the SIDs in SDB_{new}. Therefore, we can use Algorithm 20 to correlate child and parent SIDs from respective write-PDTs, and pass the resulting (FSID, LSID) pairs to JIS.TestAndSetMinForeignSid in SDB_{new} at time $commit(CP)$. We update min-max indices of SDB_{new} at this same moment, but now using (SID, value) for every insert and modify update from those write-PDTs.

This approach does not work for the scenarios of tx_3 and tx_4, as they are still running at $commit(CP)$. Transaction tx_3 started during the checkpoint, which means that the SID image of its trans-PDT updates is either equal to SDB_{new} or newer. The latter is the case if anything committed between $start(CP)$ and $start(tx_3)$. Note, however, that if anything committed in this time frame, it is exactly what is contained in tx_3's write-PDTs at $start(tx_3)$. At $commit(tx_3)$, we can therefore extract update SIDs from committing trans-PDTs, and use WPDT.RidToSid (i.e. Algorithm 10) to convert those trans-PDT SIDs to write-PDT SIDs, which are compatible with SDB_{new} and can be used to perform both min-max and JIS maintenance.

Note that this solution assumes that committing trans-PDTs are still non-serialized, i.e. they are not "rolled forward" yet. We could employ an alternate strategy, in which we first serialize committing trans-PDTs with trans-PDTs that committed between $start(tx_3)$ and $commit(tx3)$, which makes them consecutive to the global write-PDTs of time $commit(tx3)$. The advantage of such an approach is that, in case serialization fails due to detection of a conflict, we do not needlessly pollute the min-max and JIS indices in SDB_{new}. Furthermore, this approach is also applicable to our fourth case, tx_4, making it the preferred strategy. After serializing committing trans-PDTs at $commit(tx_3)$ or $commit(tx_4)$, the update SIDs in those trans-PDTs have become RIDs in the current global database state. Converting those RIDs through the global write- and read-PDTs at commit (using Table.RidToSid, Algorithm 11), we end up

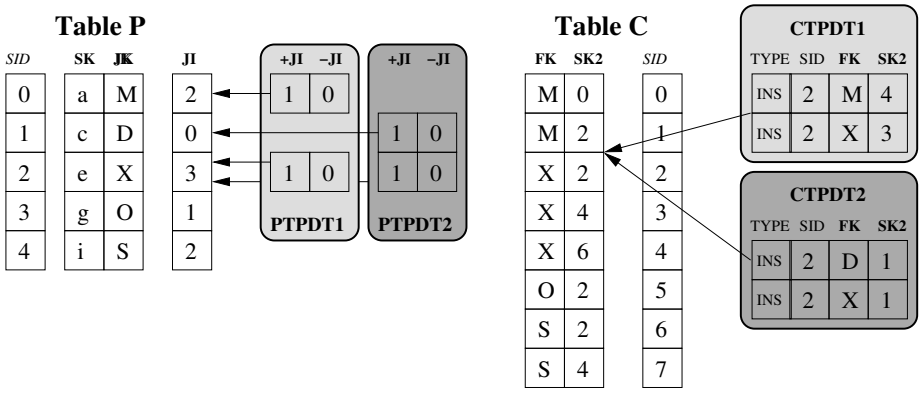


Figure 7.6: Concurrent join index updates

with SIDs that are in the SID domain of SDB_{new}, which can be used to update both min-max and JIS.

The strategy for tx_3 and tx_4 assumes trans-PDT serialization (i.e. Algorithm 8) to work. However, for child PDTs in a join index association, such serialization introduces additional challenges, which we discuss next.

7.6.3 Serializing Child-PDT Inserts

When serializing a committing trans-PDT (TPDT2) with an earlier committed trans-PDT (TPDT1), we resolve an INS-INS conflict, where both transactions insert a tuple at the same SID, by comparing and ordering those conflicting tuples on their sort key attribute values (see Serialize, Algorithm 8, line 14). Child tables in a clustered join index association, however, have their primary sort key attribute(s) in the respective parent table. Therefore, relevant sort key attribute values are not present in a child-side trans-PDT, and often not even in the corresponding parent-side trans-PDT, which complicates this conflict resolution mechanism.

Figure 7.6 illustrates the problem. It represents two transactions, tx_1 and tx_2, with conflicting child-side PDT inserts, respectively in CTPDT1 and CTPDT2. Both child PDTs insert (twice) at conflicting SID=2. The relevant changes to the parent-side PDTs are shown in PTPDT1 and PTPDT2, where, for simplicity, we only display +JI and -JI changes, together with a pointer to the stable JI count they apply to. During serialization of child-side trans-PDTs, we use this parent-side information, in conjunction with our earlier JiIncrementsIterator (Algorithm 20), to determine correct ordering of conflicting inserts, rather than comparing their (P.SK, C.SK2) sort key values. Sequentially walking through the +JI increments in both PTPDTs, and correlating them with their respective CTPDT inserts, we can easily see that the ('M', 4) tuple from tx_1, which references parent tuple at SID=0, comes before ('D', 1) from tx_2, which references the next parent tuple at SID=1. The ('X', 3) and ('X', 1) child inserts refer to the same parent tuple, as indicated by a +JI count of 1 for the parent tuple at SID=2 in both PTPDTs. Therefore, this conflict should be resolved by looking at C.SK2, putting ('X', 1) before ('X', 3).

To come to a formal procedure, assume that one of the transactions, tx_1,

	PTPDT1 INS	PTPDT1 MJI
PTPDT2 INS	1	1
PTPDT2 MJI	*-1*	0

Table 7.1: Comparison matrix for same position updates (either insert (INS) or modify join index (MJI)) in committing PTPDT2 and committed PTPDT1.

committed successfully while tx_2 was still running. When tx_2 tries to commit next, CTPDT2 is still *aligned* with CTPDT1. To be able to commit CTPDT2, we have to make it *consecutive* to the already committed CTPDT1 by serializing it. This means that we have to transpose the insert SIDs in CTPDT2 so that they are relative to the table image after merging updates from CTPDT1. To accomplish this, we need to determine where the inserts in CTPDT2 will end up with respect to those in CTPDT1. We can not deduce this by looking at the child inserts only, as they do not contain complete sort attribute information. All we have is the foreign key field, C.FK, which only gives us information about tuple *clustering*. To resolve actual ordering, we could compare the P.SK values of the parent tuple each child insert refers to.

Rather than comparing P.SK values, however, we can also use the virtual C.FRID column, as parent-side RIDs enumerate parent tuples precisely in sort key ordering. Given that a parent- and child-side trans-PDT from the same transaction are guaranteed to contain updates from the same time interval, we can use the JiIncrementsIterator from Algorithm 20 to compute the FRIDs and FSIDs for each insert we encounter during serialization of a CTPDT. We extend Serialize (Algorithm 8), which iterates over two trans-PDTs, a *committed* and a *committing* one, to employ two JiIncrementsIterator objects, $committedParentIter$ over the already committed (PTPDT1, CTPDT1), and $committingParentIter$ over (PTPDT2, CTPDT2). Here, PTPDT2 should already have been serialized against PTPDT1, to guarantee that it is free of conflicts, and that a correct ordering of parent inserts has already been established in case of INS-INS conflicts.

The $committedParentIter$ and $committingParentIter$ are advanced in lockstep with Serialize's iterations over both CTPDTs, using calls to committedParentIter.nextChildInsert() or committingParentIter.nextChildInsert() *after* consuming a child-side insert from CTPDT1 or CTPDT2 respectively. The result is, that for every CTPDT insert we encounter, the corresponding JiIncrementsIterator always refers to the matching PTPDT tuple. Whenever Serialize detects a pair of SID-conflicting inserts, rather than relying on a simple value based SK comparison, we employ the Compare routine from Algorithm 21.

Algorithm 21 uses positional information from $committedParentIter$ and $committingParentIter$ to determine the relative ordering of the tuples under their cursor. Because we assume PTPDT2 to be consecutive to PTPDT1 (i.e. it has already been serialized), we compare SIDs from PTPDT2 with RIDs from PTPDT1. If these differ, resolving the comparison is trivial. If, however, the PTPDT2 SID is equal to the PTPDT1 RID, we enter the more complex logic at line 9.

This logic is based on the comparison matrix in Table 7.1. Recall that the parent-side JiIncrementsIterator only inspects tuples with a non-zero +JI field, which can only be insert (INS) or modify join index (MJI) updates. With the

Algorithm 21 Compare(*committedParentIter*, *committingParentIter*)
Given two JiIncrementsIterator objects over two consecutive trans-PDTs from the same parent table, this routine gives an answer to the question how the tuples pointed to by each iterator compare (in terms of sort key order). If iterator *committedParentIter* indexes the smaller tuple, we return -1, 1 if larger, and 0 in case of equality.

Require: *committingParentIter* consecutive to *committedParentIter*.
1: $committedRid \leftarrow committedParentIter.curFRID$
2: $committingSid \leftarrow committingParentIter.curFSID$
3: $committedType \leftarrow committedParentIter.getCurUpdateType()$
4: $committingType \leftarrow committingParentIter.getCurUpdateType()$
5: **if** $committedRid < committingSid$ **then**
6: **return** -1
7: **else if** $committedRid > committingSid$ **then**
8: **return** 1
9: **else** {Position does not give resolution}
10: **if** $committingType \equiv INS$ **then** {Insert goes before existing tuple}
11: **return** 1
12: **else if** $committedType \equiv INS$ **then** {Committed Insert goes first}
13: **return** -1
14: **else** {Same-SID MJI updates, resolve using child-side SK}
15: **return** 0
16: **end if**
17: **end if**

SID domain of PTPDT2 being consecutive to PTPDT1, an insert SID from PTPDT2 indicates that the new tuple should go *before* the existing tuple at the equal PTPDT1 RID. This means that a PTPDT2 insert always compares smaller, so we return 1. In case we do not have an insert in PTPDT2, it must be a MJI update of the join index column. Such an MJI, due to isolation, can never be with respect to an insert that got committed while the committing transaction was still running. Therefore, if PTPDT1 points to an INS, that insert goes before the stable tuple being modified by the MJI in PTPDT2, so that the MJI compares to be the bigger value, and we return -1. If, on the other hand, PTPDT1 also points to an MJI update, both refer to the same underlying tuple, and we return 0 to indicate equal comparison. In this latter scenario, we might still be able to resolve the conflict by comparing secondary sort key attributes in the child table (i.e. C.SK2 in our example). If these do not exist, or compare equal as well, we have a real sort key conflict, and should abort.

Example

We end this discussion by walking through the more complex example from Figure 7.7, where we assume tx_1 to be committed, and tx_2 is about to commit, but has already serialized PTPDT2. PTPDT2 is shown to the right of PTPDT1, to indicate that it is consecutive to the result of merging Table P and PTPDT1. CTPDT1 and CTPDT2 are drawn below each other, to indicate that they are still both consecutive to the SID image of Table C, i.e. they are aligned. If

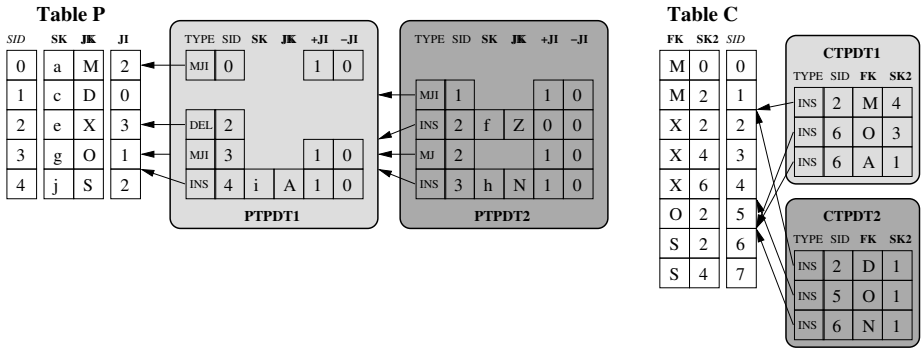


Figure 7.7: More complex concurrent join index updates

Step	CTPDT1 entry	CTPDT2 (entry, SID_{new})	PTPDT1 (SID, δ, RID)	PTPDT2 (SID)	Compare
1	2,('M',4)	2,('D',1), -	0,0,0	1	-1
2	6,('O',3)	2,('D',1), 3	3,-1,2	1	-
3	6,('O',3)	5,('O',1), 6	3,-1,2	2	-
4	6,('O',3)	6,('N',1), -	3,-1,2	3	-1
5	6,('A',1)	6,('N',1), 8	4,-1,3	3	1
6	6,('A',1)	-	4,-1,3	-	-

Table 7.2: Serialization steps for CTPDT2 from Figure 7.7. The CTPDT2 column shows both the current SID and the transposed SID_{new}, which is assigned only on steps where the CTPDT2 cursor is advanced. Note that only steps 1, 4 and 5 require Compare, as those are the only steps with a SID conflict between CTPDT1 and CTPDT2.

serialization succeeds, CTPDT2 will be consecutive to the result of merging Table C with CTPDT1.

We start out iterating CTPDT1 and CTPDT2, for both of which we maintain a JiIncrementsIterator, *committedIter* and *committingIter* respectively. The *committedIter* points to the first row in PTPDT1 initially, as this contains a 'modify join index' (MJI) type update, with a non-zero +JI. The *committingIter* points to the first row in PTPDT2, again a join index modification, but with a larger SID. The child side iteration starts at the top rows of both child PDTs. This initial state is summarized in Step 1 of Table 7.2. The table shows the state of both child cursors and both parent cursors as they gradually progress through the serialization process.

On the first row, Step 1, we see that both CTPDT cursors point to an insert at SID=2 (all example CTPDT updates are inserts, so we leave out the update type). The corresponding PTPDT1 RID=0 compares smaller than the PTPDT2 SID=1, which results in Compare returning -1, as indicated in the final column. After this, both PTPDT1 and CTPDT1 cursors advance, using the lock-step JiIncrementsIterator. Note that the PTPDT1 cursor forwards to the next non-zero +JI bucket. Therefore, it skips the DEL at SID=2, which contributes a δ of -1.

In Step 2, the CTPDT SIDs are not equal, so no need to resolve a conflict with Compare. The smaller SID simply goes next in sequence, which

is SID=2 from CTPDT2. Before we advance the CTPDT2 cursor, we increment the SID=2 currently under the cursor to SID_{new}=3, to reflect the serialization against the CTPDT1 insert from Step 1. Using JiIncrementsIterator.nextChildInsert(), the PTPDT2 cursor advances along to SID=2. Note, however, that this jumps *over* the PTPDT2 INS at SID=2, as that has a zero +JI count. The next step, Step 3, is similar to Step 2, as CTPDT2 SID=5 still compares smaller than CTPDT1 SID=6, so that we again advance the PTPDT2 and CTPDT2 cursors (to their final entries), taking care to serialize SID=5 into its shifted value of SID_{new}=6 in CTPDT2.

At Step 4, we again reach a conflict, as both CTPDT cursors point at SID=6. Comparing PTPDT1 RID=2 and PTPDT2 SID=3, the PTPDT1 insert goes first. Therefore, we advance both PTPDT1 and CTPDT1 (now also to their final entries), which brings us to Step 5. In Step 5, we again have a conflict, still at SID=6. This time, however, PTPDT1 RID=3 compares equal to PTPDT2 SID=3. Both PTPDT1 and PTPDT2 cursors point to an INS update, which, by definition of the way we treat inserts, means that the PTPDT2 INS goes *before* the PTPDT1 INS. Therefore, Compare returns 1 (at line 11), so that we advance PTPDT2 and CTPDT2 cursors. This time, however, we serialize SID=6 into SID_{new}=8, i.e. a shift by two, to accommodate for the two PTPDT1 inserts that we processed up to this point. In Step 6, we are done, as we reached the end of CTPDT2, which is the one being serialized.

7.7 Experiments and Optimizations

The aim of this section is to evaluate Vectorwise's performance under heavy update loads, i.e. "stress tests", pushing PDT memory consumption to its limits. The result is a workload where update queries, background checkpoints, and regular analytic queries compete for resources. For this benchmark, we perform repeated TPC-H SF=100 power and throughput runs, under various configurations and optimizations, making a case for a *clustered* (i.e. with clustered join indices) rather than a *non-clustered* (heap) table layout. We also discuss performance bottlenecks we encounter, and analyze the impact of optimizations that aim to alleviate those bottlenecks.

7.7.1 Setup

All following experiments were conducted on a machine with two Intel Xeon E5-2650 CPUs, each with eight physical (and hyper-threaded) cores, resulting in a total of 16 physical and 32 virtual cores. The machine contains 256GB RAM and a three hard-disk raid-5. Fedora 20 (Linux 3.12.10-300.fc20.x86_64) was used as the operating system.

The TPC-H benchmark mimics a data warehouse that keeps an inventory of items and their order history. It defines 22 (read-only) analytic queries, and two *refresh functions* (RF), which periodically update the database by either adding a batch of new orders (RF1) or deleting a batch of existing orders (RF2). The refresh functions modify the two largest tables, *orders* and *lineitem*, where an order consists of multiple line-items, implemented by means of a foreign-key reference from lineitems to their respective order. In a *power run* the 22 analytic queries are executed sequentially, followed by a single batch of both

RF1 and RF2. A *throughput run*, on the other hand, executes N concurrent query streams, each of which runs the 22 analytic queries in a distinct order, and a separate, concurrent refresh stream, which performs N batches of RF1 and RF2 each. In case of our scale-factor 100 benchmark, N equals 5.

The experiments in the following sections consist of 30 TPC-H iterations, with each iteration first performing a TPC-H power run, followed by a throughput run. To ease comparison of power runs, we deviated slightly from the specification [7] by always executing the 22 analytic queries in the same order, followed by RF1 and RF2. A single batch of RF1 or RF2 respectively inserts or deletes 0.1% into/from the lineitem and orders tables. For our SF=100 setup, this boils down to (roughly) 150000 orders tuples and 600000 lineitem tuples being inserted or deleted. Furthermore, tuples that were inserted during RF1 are never the target of deletion by RF2. Over multiple iterations, total table counts therefore stay constant, keeping comparison of execution times meaningful. For PDT-based updates, this means we always delete stable tuples, never previous inserts "in-place", so that the number of PDT update entries is always in sync with the total sum of inserted and deleted tuples.

The number of concurrent streams for a throughput run is determined by the size of the data set. For our SF=100 setup, TPC-H dictates 5 query streams and 1 refresh stream. Every throughput thread executes the 22 analytic queries in a different order. The refresh stream performs 10 independent update transactions: RF1 followed by RF2, repeated 5 times, with each RF using fresh data and performing its own commit [8]. Overall, each throughput run therefore touches around 1% of tuples in both orders and lineitem tables.

The raw SF-100 data consumes around 100GB. The Vectorwise server was configured to use 16-way intra-query parallelism and to employ automatically calibrated compression. This results in a database of roughly 30GB. For runs with checkpointing enabled, a background checkpoint is triggered automatically once a table reaches an update ratio of 2%. For the 128GB main-memory setup, 32GB is dedicated to the buffer pool, and 64GB for auxiliary memory allocations, up to half of which can be claimed for PDT updates. A 256GB setup doubles these numbers to 64GB buffer pool and 128GB auxiliary. Unless stated otherwise, a 128GB configuration is used as the default.

7.7.2 Explanation of Graphs

Each graph shows 30 iterations along the x-axis. However, each iteration shows an average of five repetitions. The left y-axis corresponds to the histogram bars and shows time in seconds. Each bar is subdivided into time spent on read-only analytic queries, RF1 execution, RF1 commit, RF2 execution and RF2 commit. The digit within each bar shows the number of repetitions (out of 5) that had either partial or full overlap with a background checkpoint. The number can differ from 0 or 5 as start and completion times of a checkpoint are not deterministic. The right y-axis holds the scale for the line plot, which represents PDT memory consumption in gigabytes.

[7] Officially, query execution order should change every power run.

[8] These commits put extra stress on the Vectorwise server, as the TPC-H specification leaves the transaction boundaries to be chosen freely. I.e. a single commit after all 10 refresh batches would be allowed as well.

7.7.3 Non-Clustered Baseline

The results in this section were obtained using a non-clustered table layout, where each table is an unordered heap, no join indices are created to connect tables related by a foreign-key, and hash-join is typically used to evaluate queries with joins.

Without Checkpointing

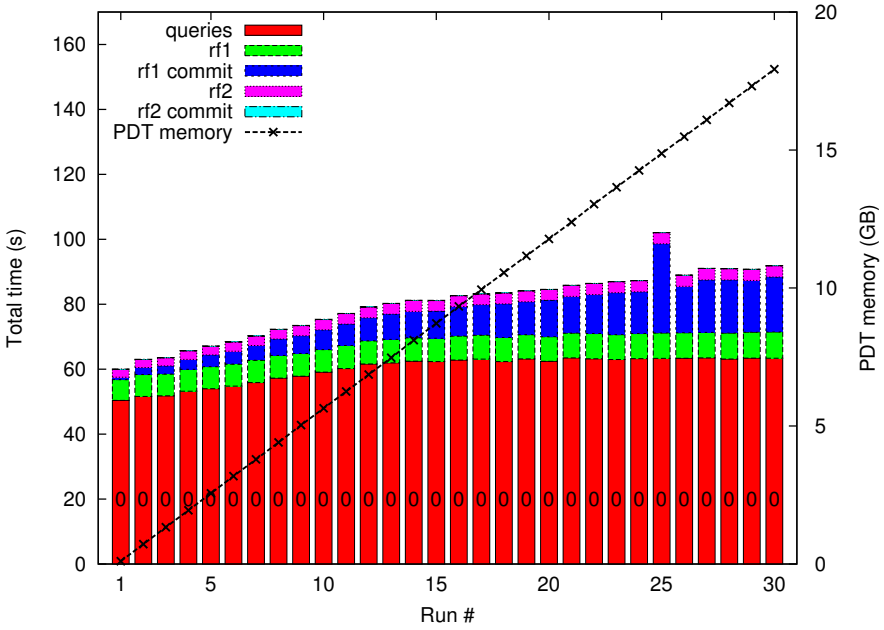
Figure 7.8a shows initial results for power runs on a non-clustered (i.e. heap) table layout, without checkpointing. The graph clearly shows that query times keep rising until run 14, and that commit times, most notably for RF1, keep increasing consistently. We also see an outlier in run 25. Ignoring this outlier, total times (i.e. including RF times) for a power run degrade from 60s in the first run to 92s in run 30. During this time, PDT memory consumption grows from 0 to roughly 18GB. Note, however, that most of those updates come from the throughput runs, that are run immediately after each power run. A single power run adds 0.1% inserts and also deletes 0.1%, which results in roughly 105MB of PDT update data.

The read-only query times (i.e. queries 1-22, ignoring RF functions) increase from 51s to 64s over the first 14 runs, at which they flatten out. This is caused by PDT inserts ending up at the tail of the unordered orders and lineitem tables, introducing skew in the distribution of PDT updates over parallel threads (which use range partitioning). In general, the higher the ratio of updates within the output vector of a scan, the worse the performance, as merging of PDT inserts requires logic that resembles tuple-at-a-time processing. In this scenario, where inserts end up in the final parallel partitions (based on range partitioning), scan performance degrades from 0.7 cycles per tuple to 2.5. Once the tail partition is saturated with PDT updates, around run 14, further degradation of query times stops. PDT memory consumption, however, keeps growing linearly, as indicated by the line plot (with scale on the right y-axis). This is caused by checkpointing being disabled for this experiment.

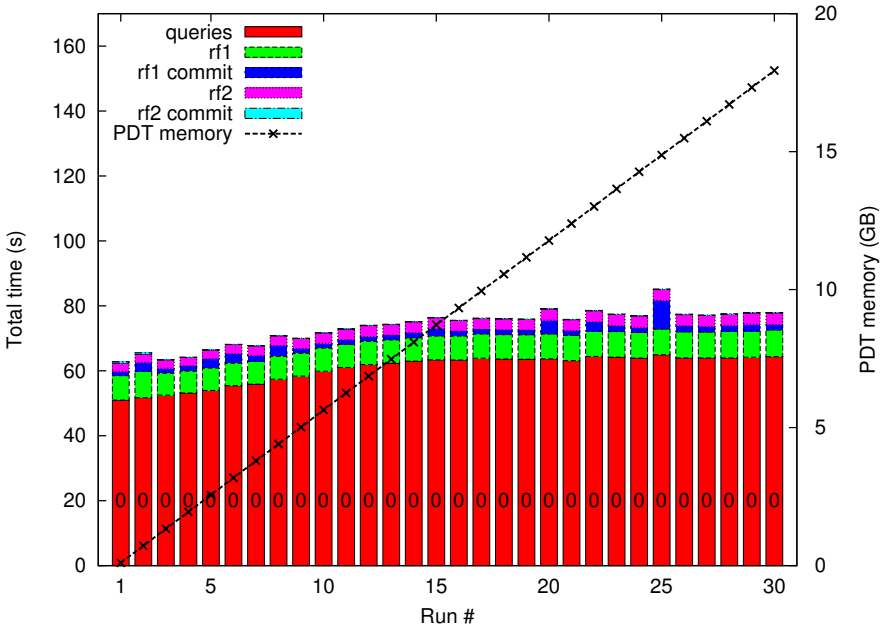
Figure 7.8a also shows a considerable degradation of transaction commit times for RF1. This can be attributed to two factors: write-PDT to read-PDT propagation on every commit, and increasing costs for generating snapshot copies of the growing read-PDTs. Both issues are relatively easy to optimize.

The initial default behavior of Vectorwise was to trigger write-to-read propagation once the write-PDT exceeds a configurable, fixed threshold. In this experiment, this threshold was set to 4MB (i.e. in the order of the cache size), which happened to get exceeded on every commit. Write-to-read propagation is relatively expensive, as it generates a snapshot copy of the read-PDT, so that write-PDT updates can be migrated into an isolated copy. We concluded that reducing the number of write-to-read propagations is more important than keeping the write-PDT strictly in the cache, and changed the trigger to be based on a dynamic threshold, i.e. a fraction (f) of the current read-PDT size. The effect is that, assuming fixed size commits, write-to-read propagation is (eventually) only called once every $1/f$'th commit.

Besides the number of read-PDT snapshot copies made, the copying process itself turned out to be expensive as well. Not only does it copy the PDT tree structure, but also the full value-space (i.e. all insert tuple data). Observing



(a) Baseline



(b) Optimized commit

Figure 7.8: Non-clustered TPC-H power runs without automatic checkpointing.

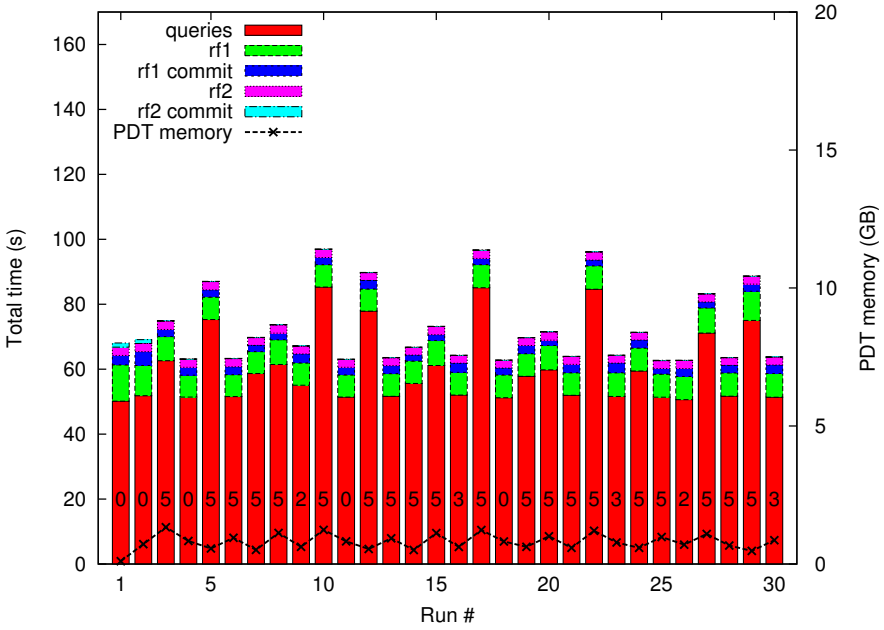
the difference between commit times for RF1 and RF2, which holds only deletes that do not store anything in value-space, we can even conclude that copying and migrating value-space data constitutes the bulk of the RF1 commit cost. Copying the PDT tree structure makes sense, as it is relatively small and subject to scattered changes. Snapshot-copying the value-space data, however, is often needless, as this is an append-only data structure, stored in pre-allocated memory buffers. These buffers are subject to re-sizing, and often end up with unused space in the tail. Therefore, if the write-PDT value-space data fits in the unused memory area of the read-PDT value-space, we simply append the write-PDT data directly into the global read-PDT, avoiding a snapshot copy of the value-space part of the read-PDT. This is safe as long as only one commit can be active at any moment in time, which is the situation in Vectorwise.

After implementing both optimizations, using a write-to-read propagation threshold of $0.1 \times size$(read-PDT), RF1 commit times improve considerably, as shown in Figure 7.8b. The only exception is run 25, where we see a spike that can be attributed to increased RF1 commit times. Recall that each bar in the graph represents the average of 5 runs. Closer inspection of server logs showed that the spikes are actually being caused by an outlier in only one of those 5 runs, where RF1 commit suddenly takes 90s instead of the 20-25s observed for the remaining 4 runs (in the baseline experiment). The additional time is spent migrating write-PDT value-space data to the read-PDT. All memory allocations were verified to be properly aligned and the issue failed to occur with profiling enabled. The most likely explanation therefore remains that it has to do with an oddity around memory copies on the NUMA architecture, maybe in combination with Vectorwise's own memory manager.

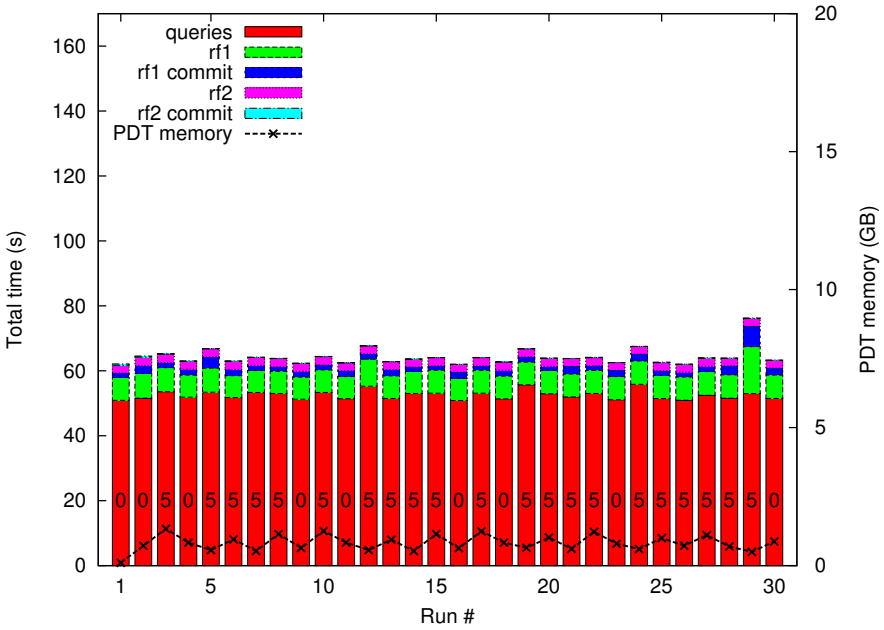
With Checkpointing

To avoid PDT memory consumption from growing out of bounds, we enable checkpointing to rebuild tables in the background. Figure 7.9a shows the effect of enabling automatic checkpoints, which results in both orders and lineitem tables being rebuilt independently of each other once they reach an update ratio of 2%. We can clearly observe the positive effect of those checkpoints on PDT memory consumption, which now fluctuates around a constant average value. However, we also see regular spikes in execution times, which were not present in earlier runs without checkpointing. Inspection of logs and profiles revealed that the higher peaks (i.e. runs 5, 10, 12, 17, 22 and 29 happen to overlap with the final half of the time-frame where a background checkpoint of the (larger) lineitem table is running and committing. After the commit of such a checkpoint, queries suffer from I/O misses, as the checkpointing process of the large lineitem table evicted some blocks from the buffer manager, while the compressed TPC-H data-set just about fits the 32GB buffer manager space. Doubling the buffer manager memory makes those I/O misses disappear, as illustrated by the benchmark results in Figure 7.9b, where the impact of a background checkpoint is reduced to a slight CPU overhead in the form of a single thread that scans and recompresses a checkpointed table.

The overall trend of both graphs is that, on average, query times, commit times and PDT memory consumption all stay flat. Checkpointing successfully prevents the amount of PDT updates in memory from growing out of bounds, thereby reducing both the negative effect of PDT updates on scan performance



(a) 128GB memory configuration



(b) 256GB memory configuration (in-memory checkpointing)

Figure 7.9: Non-clustered TPC-H power runs with automatic checkpointing.

and expensive PDT snapshotting times during commit.

7.7.4 Clustered Table Layout

In this section, we show TPC-H results on a clustered table layout, where tables with a foreign key may be ordered on the sort key of the referenced parent table and connected by means of a join index. This allows clustering of data to be exploited, which results in reduced scan ranges (and data volume) that can be propagated over to connected tables. Besides, clustering allows merge-join to be used for evaluating plans with joins, which is typically less CPU and memory intensive than the typical hash-join used on an unordered heap table.

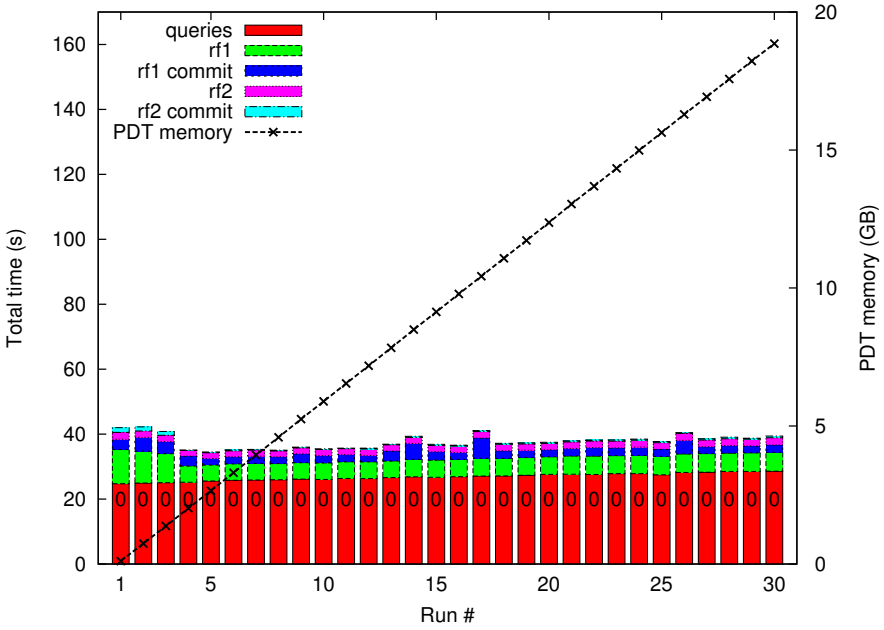
Figure 7.10a depicts clustered results without automatic checkpointing, and shows a clear improvement in execution time over non-clustered runs. The read-only query times improve by roughly a factor 2, degrading from 25s in run 1 to 28s in run 30, and do not exhibit the skew where PDT updates end up in the tail parallel partition(s), which caused non-clustered runs to degrade much worse, from 51s to 64s. In terms of total time, including refresh routines, the benefit becomes slightly less than a factor two, 35-40s for clustered, versus 63-79s (ignoring the outlier in run 25) for non-clustered, as both layouts require similar time to perform both RF1 and RF2, around 12s on average. Still, we can conclude that a clustered layout shows significant benefits over a non-clustered layout in terms of query evaluation performance, without introducing negative effects in terms of update performance.

With automatic checkpointing enabled, Figure 7.10b first illustrates what happens without the deferred index maintenance support described in Section 7.6.2. What we see is that runs 3 to 22 are continuously overlapped by a background checkpoint, but that PDT memory consumption keeps rising. In fact, we are dealing with four background checkpoints rather than one. However, each of those checkpoints fails to commit, due to concurrent updates to the join index summary (JIS), which used to result in a conflict. The initial policy of Vectorwise was to abort the checkpointing transaction upon such a conflict, retrying up to three times. Clearly, all subsequent efforts fail as well, resulting in wasted resources only.

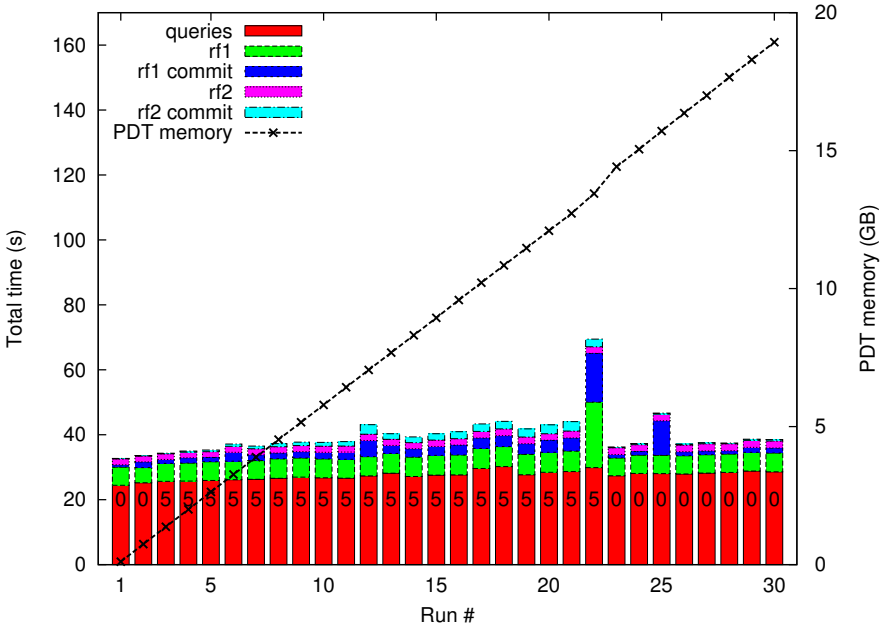
With the deferred JIS maintenance from Section 7.6.2 in place, PDT inserts into the lineitem table that occurred during a background checkpoint are properly applied to the newly checkpointed image once the checkpoint commits, eliminating the need to abort with a conflict. After adding this functionality, the drops in PDT memory consumption in Figure 7.11a indicate that checkpoints commit successfully now. Overall, however, PDT memory consumption stays considerably higher than in the non-clustered scenario. The reason is twofold.

First of all, rebuilding a clustered layout is more complex and expensive, as updates are scattered over the item and lineitem tables. Therefore, both tables have to be rebuilt in their entirety. Also, connected tables within a cluster need to be rebuilt in one go, using merge-joins to rebuild the join indices between parent and child tables. Non-clustered heap tables, on the other hand, have most updates in the tail, and can be checkpointed independently of each other.

The second reason for higher PDT memory consumption is that, while a checkpoint takes longer to complete, the arrival rate of new updates during such a checkpoint is about a factor two higher, simply because query throughput on the clustered layout is roughly twice as high. This could be remedied by using

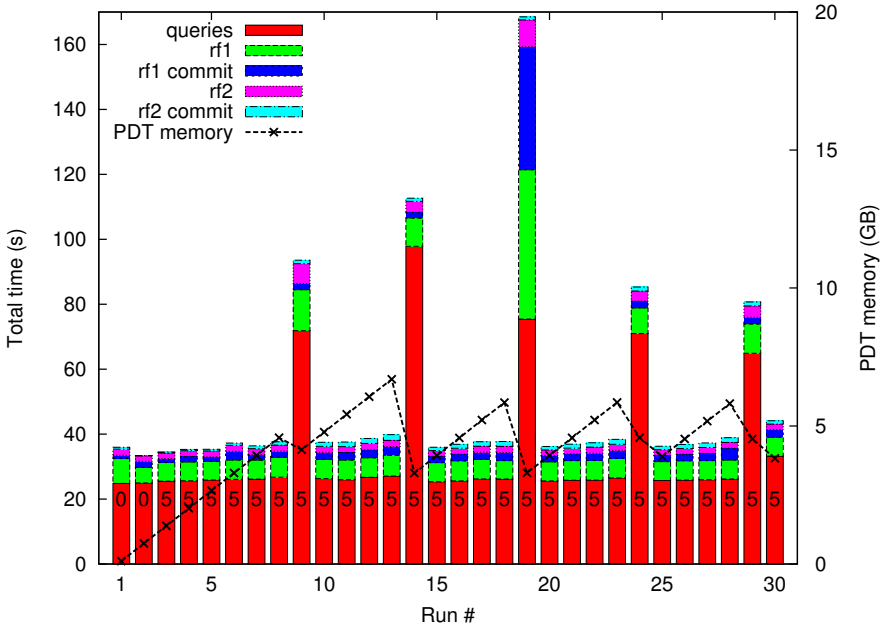


(a) Without automatic checkpointing.

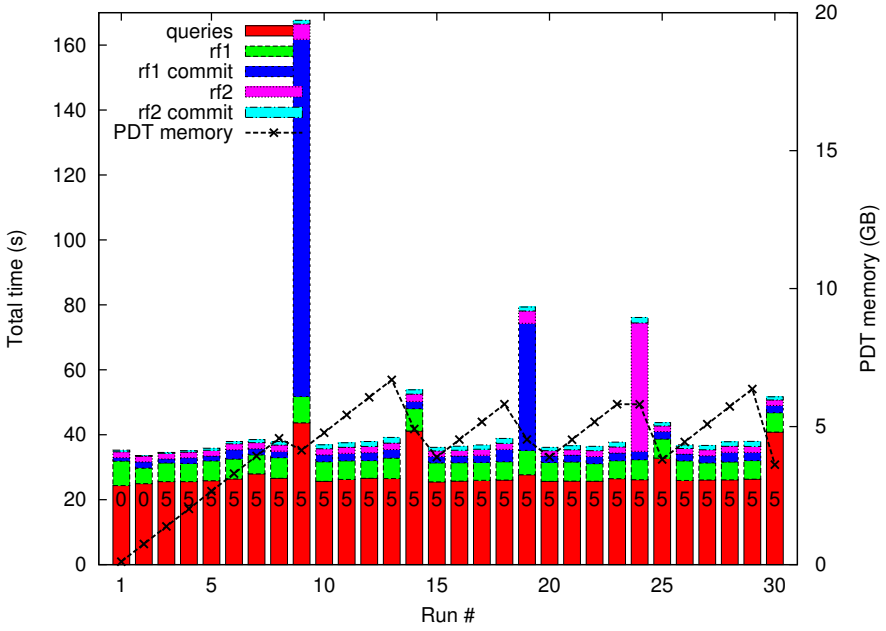


(b) Automatic checkpointing fails to commit

Figure 7.10: Clustered TPC-H power runs.



(a) 128GB memory configuration



(b) 256GB memory configuration (in-memory checkpointing)

Figure 7.11: Clustered TPC-H power runs with automatic checkpointing.

a storage subsystem that is more in balance with the processing power of our 16-core 256GB NUMA machine. The three-way HDD RAID in this platform is rather unfortunate, as reading and rebuilding the TPC-H database consumes around 5 minutes I/O-wise.

Poor I/O performance is not the only reason for slow clustered checkpointing. In Figure 7.11a we also note several high spikes in execution time. These spikes correspond to runs that overlap a *committing* background checkpoint, after which we switch to the new table images. A fraction of those spikes can be attributed to an increase in the read-only "queries" component. As in the non-clustered case, this is caused by I/O misses due to evicted blocks after the switch. This time, however, doubling the server memory does eliminate most of the I/O misses, as indicated by the queries component in the bars of Figure 7.11b, but fails to eliminate the spikes entirely.

Analysis of execution profiles revealed that the remaining spikes are caused by checkpoint commit taking excessively long, often around 270-370 seconds. These commits reuse Vectorwise's general transaction commit routine, which is guarded by a global mutex. Therefore, this mutex is also used to start, abort, or commit a regular transaction, meaning that, as long as a checkpoint (or regular transaction) is committing, no other transaction can start or finish. We will not go into much technical details on this, but a brief discussion of where time is spent during checkpoint commit aids in reaching our final conclusion.

In our clustered benchmarks, a typical checkpoint commit iterates a list of roughly 150 trans-PDTs, from transactions that committed during the checkpoints lifetime. Each refresh function generates two trans-PDTs, one for orders, one for lineitem. So, overall, we are dealing with 75 transactions, containing around 4GB of PDT data, the bulk of which is consumed by roughly 22 million lineitem inserts and 5.5 million inserts into orders. First of all, these PDTs are iterated in commit order, and propagated into two fresh (write-) PDTs, which eventually become the initial read-PDTs in the newly checkpointed image. As these propagations reuse the trans-to-write propagation of regular transaction commits, a copy-on-write snapshotting mechanism was triggered on every propagation into this ever growing write-PDT. Given that we did not switch to the checkpointed image yet, this write-PDT is still free from concurrent readers, making this snapshotting a needless waste of CPU time.

Next, both these initial read-PDTs are used to perform deferred index maintenance for the orders and lineitem tables. This involves iterating over the inserts in these PDTs and calling min-max and JIS maintenance routines. Especially the min-max update routine turned out to be highly inefficient for such bulk updates, involving a function call per update (i.e. per PDT insert tuple), where each tuple triggers two "compare" calls per attribute, to compare against the current min and max values. Overall, this took more than a minute to handle our 27.5 million insert updates. One can easily envision a "bulk" algorithm that should be capable to perform these updates in a few seconds. For example, given that the min-max SID-based partitioning is known, one could turn the process around: passing SID ranges to the PDT and letting it return min and max attribute values for the inserts found in each SID range. This could even be parallelized based on the SID partitioning.

Finally, the entire 4GB PDT data, together with the catalog changes for newly rebuilt tables, are being written to the WAL. Such "object serialization" involves a lot of byte-level work, and is therefore very slow, taking around 70

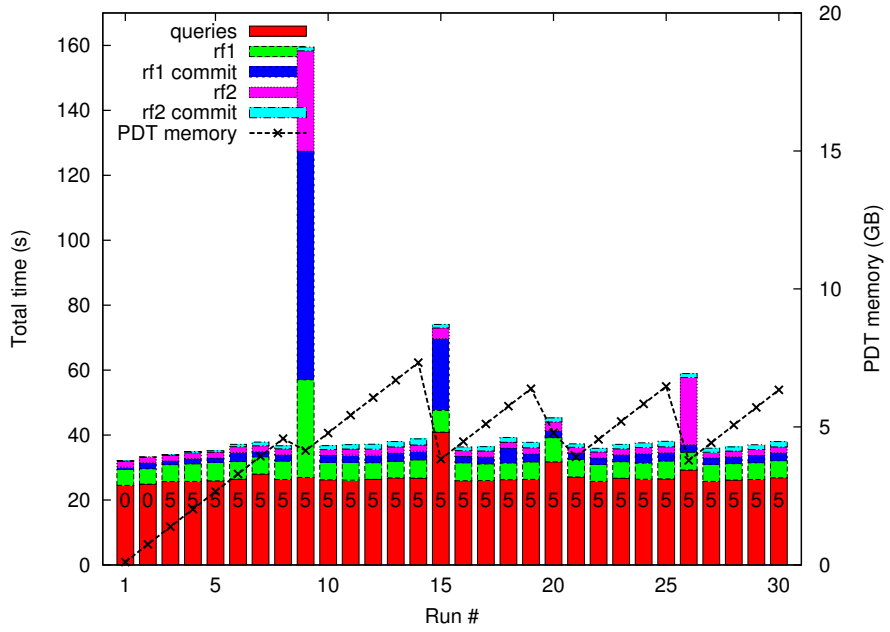


Figure 7.12: Clustered TPC-H power runs with improved checkpoint commit times.

seconds. If checkpointing starts a fresh, private WAL, there is no need to do such serialization while holding the global mutex. If checkpointing updates the current WAL, there should not even be a need to serialize the PDT data, as that has already been done when the transactions committed originally. This logic should be carefully analyzed and improved within Vectorwise.

To conclude, most of the checkpoint commit work can be further optimized and should be moved outside the commit mutex wherever possible. This involves a significant overhaul of the commit infrastructure, which, due to limited time and research interest, is considered future work for Vectorwise. Disabling the copy-on-write snapshotting for the newly built "checkpoint" PDTs is the only issue that could be improved with relative ease. The final results with that optimization in place are shown in Figure 7.12. With a properly optimized implementation of checkpoint commit, however, (in-memory) results with checkpointing enabled should match, or even beat, the results without checkpointing from Figure 7.10a.

7.7.5 Throughput and Total Times

Until now we have only focused on the power runs of our TPC-H stress tests. After each power run, however, we also ran the throughput test, with five concurrent query streams and a refresh stream consisting of a five times heavier update load. The graphs for these runs are much less informative, as these do not have detailed profiling information. Therefore, we only show the final results, with checkpointing enabled, for both the non-clustered and clustered scenarios, in Figures 7.13a and 7.13b, respectively. The total length of each bar

Configuration	non-clustered	clustered
without checkpointing	8771	4989
with checkpointing	8293	4880

Table 7.3: Total execution times of 30 SF-100 TPC-H power and throughput runs

represents the average query-stream completion time, while the RF component of each bar represents completion time for the single refresh thread.

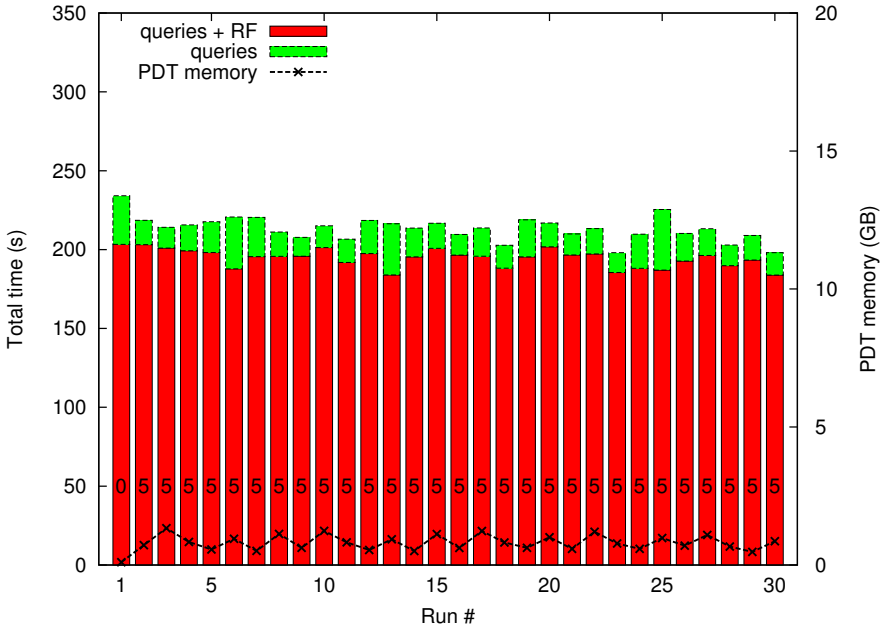
As with the power results, we see a difference of almost a factor 2 improvement of clustered over non-clustered. Due to the concurrent workload, however, a single query stream takes around four times as long to complete, compared to the "stand-alone" power runs. Given the level of concurrency, with five query streams, an update stream and a background checkpoint transaction, this super-linear degradation in performance can be considered satisfactory.

Another interesting observation can be made from Table 7.3, which lists the total execution time to complete the full experiment of 30 power and throughput runs (averaged over 5 repetitions). Here, we see that experiments with checkpointing achieve higher overall throughput than without checkpointing. With a proper implementation of checkpoint commit, which should remove the spikes from our graphs, and a faster I/O subsystem, which lets checkpoints complete faster, the advantage should be even more profound. However, even given the suboptimalities, we see the advantage of running regular checkpoints, as they free up PDT memory to query processing and speed up merging of updates during scans due to a reduced update ratio. The latter especially holds for the non-clustered scenario, where we saw significant degradation of query times due to the skewed distribution of inserts over parallel partitions.

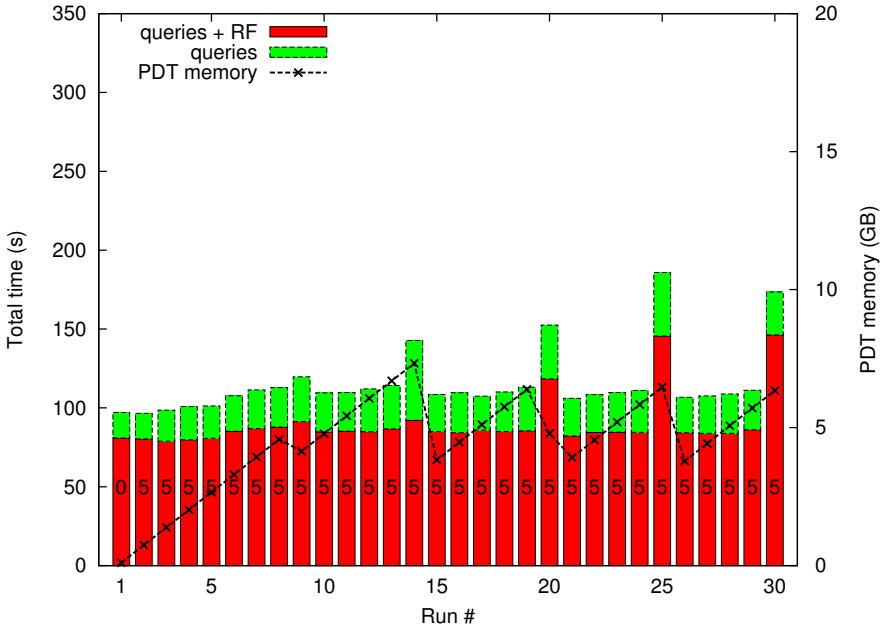
7.8 Summary and Conclusions

This chapter discussed how Vectorwise's position-based indices, the min-max index and join index, can be maintained under updates. We proposed a compressed, but updateable, representation for join indices, and discussed complications involving PDT inserts into the referencing table of a join index relationship. All these complications involve the maintenance of the auxiliary join index summary (JIS) structure, especially under concurrent update loads or during a background checkpoint. Solutions to all these problems were provided, and TPC-H benchmark results were used to illustrate the benefits of a clustered table layout, as enabled by the use of join indices.

However, support for an updateable clustered table layout severely complicates the system. The design and implementation were far from trivial, allowing subtle bugs to slip in. It is therefore a valid question whether a performance improvement of "only" a factor two justifies all this added complexity. Note, however, that this factor two was on TPC-H only, and that on other, more selective workloads, the benefits of clustering may be higher. With most of the dirty work being done, it is nice to have freedom in deciding whether the complexity added by supporting clustered tables is worth the effort in terms of implementation and testing overhead.



(a) Non-Clustered



(b) Clustered

Figure 7.13: TPC-H throughput results with automatic checkpointing.

Chapter 8

Conclusion

The following sections summarize the core contributions of this thesis and analyze their impact. They also provide angles for potential future research.

8.1 Vectorwise

By no means is this section intended to be a claim that this thesis contributes the Vectorwise DBMS in its entirety. The work presented here did, however, play a significant role in Vectorwise becoming what it is today: one of the worlds fastest analytic databases.

Vectorwise is a column-oriented database system designed and built from scratch, with high-performance analytics on large data sets as a main goal. Its *vectorized execution engine* was designed to achieve high performance on deeply pipelined, super-scalar CPUs, and written in a way that allows compilers to produce efficient, low-level code automatically. Furthermore, Vectorwise has a strong focus on data access patterns that fit the performance characteristics of a hierarchical memory layout, trying to keep as much relevant data as close to the CPU as possible.

In both these areas, the use of a *columnar storage layout* (DSM) plays an important role. Tight "vectorized" loops over arrays of attribute values provide a sequential access pattern and are easy targets for optimization by a compiler. On disk, a columnar storage layout helps reduce scan volume by only reading relevant attribute data from disk. This is important in fighting the growing discrepancy between CPU power on one hand, and data access bandwidth and latency on the other. However, even with the use of DSM and sharing of scans by concurrent queries, I/O bandwidth can become a bottleneck, given the computational power of a vectorized execution engine. Therefore, the work in this thesis first focused on utilizing compression to further improve scan performance.

To qualify as a proper DBMS, however, it should also be possible to make transactional changes to the data being stored. In a column-oriented system that is heavily optimized towards scan- and compute-intensive workloads, doing this without compromising its analytic performance is a major challenge. The second focus of this thesis was therefore on the integration of real-time, transactional updates into the Vectorwise architecture.

8.2 Light-weight Column Compression

8.2.1 Contributions

We propose to use data compression as a means to speed up I/O-bound queries, by increasing the effective disk bandwidth of column scans. By reading compressed data from disk into memory, and decompress it on-demand at a minimal cost, we can achieve a perceived I/O bandwidth that is equal to the physical I/O bandwidth times the compression ratio. Therefore, a good compression ratio is important, but it is even more important that compression algorithms operate at low CPU overhead, to leave sufficient CPU cycles for further query evaluation and minimize negative impact on CPU-bound queries in general. To achieve our goal of low-overhead decompression, we propose the use of light-weight, super-scalar algorithms, together with into-cache decompression.

Additionally, data compression can be used to increase the effective caching capacity of a buffer manager. For a fixed system configuration, this allows more data to be kept close to the CPU, improving overall system throughput.

Super-scalar algorithms. We contribute three new compression schemes, PFOR, PFOR-DELTA and PDICT, that are specifically designed for the super-scalar capabilities of modern CPUs. These algorithms are kept simple and light-weight by utilizing the knowledge that attribute values within a column are from the same domain and of equal type. Therefore, (de)compression can be implemented as CPU-efficient loops over vectors of column values. We focus on optimizing such loops by avoiding *if-then-else* constructs and *data dependencies* in loop bodies, so that compilers can produce code with sufficient instruction level parallelism (ILP) to achieve high instructions per cycle (IPC) on modern out-of-order, wide-issue CPUs. The result is that PFOR, PFOR-DELTA and PDICT compress data at more than a GB/s, and decompress a multitude of that, which makes them more than 10 times faster than other speed-tuned compression algorithms at the time. This allows them to improve I/O bandwidth even on HDD RAID and SSD drives that read and write data at rates of hundreds of MB/s.

Into-cache decompression. We propose to cache pages in the buffer manager (i.e. in RAM) in compressed form, and decompress data on the the boundary between RAM and CPU cache. This can be achieved by decompressing only fragments of a compressed page, that fit the CPU cache, in an on-demand fashion, just before the query processor needs them. By avoiding materialization of decompressed pages in RAM, we minimize main memory traffic, reducing the likelihood of memory bandwidth becoming a bottleneck.

Improved Compression Ratios PDICT and PFOR are generalizations of respectively dictionary and Frame-Of-Reference (FOR) or prefix-suppression (PS) compression, that were proposed previously [Ter, GRS98, WKHM00]. In contrast to these schemes, our compression methods can gracefully handle data distributions with outliers, allowing for better compression ratios on such data. The way we handle outliers adheres the super-scalar nature of our algorithms, as we treat them as exceptions, to be dealt with in a separate "patching" loop, outside the core decompression loop.

PFOR-DELTA is our super-scalar implementation of differential coding, also called delta coding, extended with patching for outliers. Differential coding encodes the differences, or deltas, between subsequent integer values, and is therefore well suited for compression of sorted sequences. This makes PFOR-DELTA applicable to areas other than the compression of sorted table columns, for example to compress inverted files, or postings lists, that occur in text search engines. We have shown that PFOR-DELTA compares favorably, in terms of both compression ratio and speed, to custom compression methods from the information retrieval community.

Decompression of a random, delta-coded value requires computing the sum of all preceding values, i.e. the prefix sum, which can be costly. We therefore propose the insertion of *fine-grained access points*, entry points from which we can start decompression, at fixed intervals.

8.2.2 Research Impact

Our compression work was published already some time ago, at ICDE'06. Since then, a lot of related and follow-up research has been done, not only within our research group at CWI, but also by unaffiliated database and information retrieval (IR) researchers.

Within the database community, column compression trade-offs have been researched in [HRSD07], while improvements to string dictionary compression can be found in [BHF09]. The authors of [AMF06] focus on compression aware query evaluation, i.e. operating directly on compressed data, an idea that is implemented in both SAP HANA [FML+12] and IBM's DB2 with BLU acceleration [RAB+13].

In the IR community, where fast inverted file (de)compression is being used as a mechanism to improve both query throughput and memory caching capacity [BC07], PFOR-DELTA has had a particularly strong impact, and is still being referenced as a performance baseline to beat [ZLS08, LB13]. PFOR-DELTA has also been implemented in the open-source Apache Lucene text search engine [Apac], and researched in the context of Unicorn, a graph search system from Facebook [CBB+13]. Improvements to the PFOR-DELTA algorithm are proposed in [YDS09], where NewPFD aims to get rid of compulsory exceptions, which become too expensive at lower code word sizes, while OptPFD further improves the compression ratio by adding the capability to encode each 128-entry segment with a separate, optimal code word size. Interestingly, the authors do include our original PFOR-DELTA in their experiments that show the improvement of NewPFD and OptPFD in terms of compression ratio/decompression speed trade off, but neglect to include it in experiments that measure raw decompression speed.

Luckily, the authors of [LB13] provide a very thorough overview of the state-of-the-art in high-performance list compression techniques. They evaluate all the recent proposals, and show PFOR-DELTA to still be the faster one compared to NewPFD and OptPFD. Furthermore, they propose FastPFOR and SimplePFOR as improved implementations of NewPFD and OptPFD that come closer to the speed of PFOR-DELTA, while still obtaining improved compression ratios. Most importantly, they present novel SIMD compression algorithms, one of which, SIMD-FastPFOR, is based on our original PFOR, that beat our algorithms by almost a factor two in terms of decoding speed. They also present

SIMD versions of our code to pack/unpack arbitrary bit-width code words.

Within our own group, we used the MonetDB/X100 prototype to participate in the official 2006 TREC Terabyte Track (TREC-TB), a terabyte scale text retrieval contest [BCS06]. We implemented common IR indexing, ranking and optimization techniques on top of our relational engine, and managed to score results that compete with leading, customized IR systems from that time, both in terms of efficiency and effectiveness [HZdVB06]. We showed that, by using a hardware-conscious DBMS architecture, together with our novel compression techniques, general purpose relational technology can be used to implement large-scale information retrieval tasks flexibly and effectively. For the interested reader, both the methods we used and the results of the 2006 TREC-TB contest are included as Appendix A at the end of this thesis.

A more recent effort to bring column-store DBMSs and IR research closer together can be found in [MSLdV14], where the Vectorwise DBMS is shown to compete on-par with current IR systems on the ClueWeb12 data set. Furthermore, parts of our TREC-TB work were extended and formalized into a generic *array database system*, named SRAM [CHZ+08], which provides an implementation of the *matrix framework* for IR [RTK06].

8.2.3 Future Work

Advances in hardware may shift the focus of work around compression. With CPU power improving at a higher rate than memory bandwidth, the focus may shift towards improving perceived memory bandwidth, a standpoint already advocated in [LB13] The compression techniques from this thesis were also shown to increase query throughput on memory resident workloads in combination with a multi-core CPU in [Żuk09], but more research opportunities may exist in this area.

The wide availability of high bandwidth SSD RAIDs may also call for faster decompression speeds. With CPU clock speed stagnating, and computational power becoming available mostly through additional cores, it may be beneficial to fully dedicate one or more cores to decompression only. Such an approach might also open doors for slightly heavier compression techniques, which could be interesting for compression of more complex data, like "free form" (i.e. non-dictionary) strings or floating point numbers.

8.3 Positional Differential Updates

To provide transactional update functionality on top of a scan-optimized, columnar storage, we propose the use of differential update mechanisms based on tuple positions. Differential updates, maintained in an update-friendly write-store, are widely accepted as the preferred alternative to in-place updates for column-stores [CK85, Sto05], due to the latter's I/O cost being proportional to the number of attributes. We contribute new *positional* update maintenance methods that better match the read-optimized nature of both column-oriented storage and vectorized execution.

8.3.1 Contributions

This thesis makes the following contributions towards a fully positional and transactional update infrastructure.

PDT data structure. We contribute a novel tree data structure, the *positional delta tree*, that maintains differential updates, i.e. insert/delete/modify, against an immutable "stable" table on disk. Within the leaves of its tree structure, the PDT holds update entries, organized sequentially by a *stable ID* (SID), i.e. the unique offset of a stable tuple within a stored table, which marks the position where each update entry applies to. The result is that updates encountered during a walk through the PDT leaves can be *merged* into a sequential scan of its stable table in linear time. Such a merge produces the current table image as output, in which we enumerate tuples sequentially with a unique *record ID* (RID). Due to inserts and deletes at arbitrary positions in a table, the RID of a tuple is highly volatile. However, the strength of the PDT data structure is that new updates, with respect to RID positions within the *current* table image, can be added to the PDT in logarithmic time, while still keeping them sorted on SID to retain the linear merging complexity.

Efficient positional merging. We demonstrate that positional merging of updates is more efficient than its value-based alternative, where updates are merged (and organized) by their value(s) on one or more (sort) key attributes, in terms of both I/O and CPU costs. First of all, in a column-oriented DBMS, value-based merging may introduce additional I/O, as it requires sort-key attributes to be always scanned along for identification of update positions, even if a query has no further interest in these attributes. Furthermore, value-based merging induces a higher CPU overhead, as update positions need to be determined at run-time by comparing key attributes of update tuples against those in the scanned stream. With positional merging, on the other hand, we have precomputed positions available, and can skip forward to the next update at no additional cost. Positional merging involves simple integer arithmetic only, while value-based merging may require comparison of multi-attribute keys and/or more complex data types, like strings, both of which add to the CPU cost of merging. For these reasons, we believe positional updates to be a better fit for high-performance column-oriented systems.

Stacked PDTs for effective transaction management. We propose the use of three layers of "stacked" PDTs on top of a stable table, where each PDT contains differential updates with respect to the merge output of the layer directly below. Uncommitted updates go into a top-level trans-PDT, which holds transaction local changes that are not visible to concurrently executing transactions. This way, the trans-PDT provides lock-free isolation, and avoids the need to use timestamping mechanisms, as, for example, proposed for the single, global write-store architecture of [Sto05]. Furthermore, each trans-PDT captures exactly the write-set of a transaction, which we use for efficient, positional conflict resolution (or easy rollback) at time of commit.

The bottom-most layer is a read-PDT (i.e. for "read mostly"), which is a global PDT that contains the bulk of committed updates. In between the two, we propose to add another global layer, the write-PDT (or "write-mostly"),

where transaction PDTs commit into. Such a middle layer has several advantages. First of all, the idea is that the layering of PDTs mimics the hierarchical nature of a memory layout, where the write-PDT is kept "small and fast", i.e. within the CPUs caches, while the read-PDT may become "large and slow", growing as long as main memory allows it.

When a transaction commits its trans-PDT updates into a global PDT, a snapshot copy of the destination PDT needs to be made for proper isolation. The smaller write-PDT therefore acts as a buffer layer that is kept cheap to copy, while gathering sufficient updates to amortize the costs of snapshot copying the larger read-PDT (i.e. when propagating updates from the write-PDT into the read-PDT).

Finally, the write-PDT layer also increases the level of concurrency of the entire update architecture, as it allows transactions to keep committing while a background checkpoint, which rebuilds a new version of a table from the merge of its stable data with the (locked-down) read-PDT, is running.

Compact and updateable join index. We showed that a clustered table layout, where a child table that references a parent table through a foreign key is kept sorted on some attribute in the parent table and connected through a (clustering) join index, can considerably improve query execution times. To allow such a "rigid" storage layout to still be updated efficiently, we propose the use of a single-column, compressed join index representation that can only be updated through differential PDT updates. Basically, we extend the parent table with a special (immutable) stored column, that, for each parent tuple, stores a reference count of the number of child tuples referring to it. On child side updates, this reference count can be manipulated through special "incremental modify" PDT updates, and the stored join index column is only rebuilt during a checkpoint, just like regular table columns. The result is a join index representation with all the transactional benefits of stacked PDT updates.

8.3.2 Discussion and Future Work

Although the departure from the "one size fits all" paradigm has led to increased research and industry interest into read-optimized analytic systems, there remains a strong demand for on-line transaction support within such systems, resulting in a domain often termed (near) *real-time business intelligence* (RTBI). In practice, this translates into a desire to have the freshest data and highest update performance achievable on a system that is optimized for analytics. Within this context, the work presented in this thesis shows that positional differential update techniques can be used to satisfy the requirements associated with real-time analytics. In this section, we position this work against common architectural designs in the area of column-store updates and discuss potential future improvements. We focus on Vertica [LFV+12] and Microsoft SQL Server [LCH+11, LCF+13], both of which employ compressed, column-oriented storage, in combination with a vectorized execution model, and were discussed in Section 6.9. Vertica also has its roots in an academic prototype (i.e. C-store [Sto05]) from the same year as MonetDB/X100 (2005), while SQL Server is one of Vectorwise's strongest competitors in the single-node TPC-H rankings and has a well documented architecture.

Compared to the common "hybrid" architecture, with a column-oriented, compressed read-store (RS) and an updateable write-store (WS), consisting of a row-oriented structure, INS, for new tuples (e.g. a B^+-tree), and a separate structure, DEL, for deletion markers (e.g. a bit map or deletion vector), we believe that an architecture with stacked PDTs has several benefits. First of all, PDT updates have the benefit of efficiently handling Modify (i.e. SQL UPDATE). The "standard" method of implementing Modify as a deletion of the original tuple followed by re-insertion of that tuple with one or more attributes modified, is rather wasteful, both in terms of execution time and storage. It requires *all* attributes of the tuple being modified to be retrieved from disk, which is notoriously column-store unfriendly, while most of the retrieved attribute values are added back to the write-store unmodified, clogging it up needlessly. PDTs, on the other hand, allow a Modify query to be evaluated accessing only the columns of attributes involved in the Modify query plan, and only store attribute values that were actually modified in the differential structure.

Second, updates to separate, global INS and DEL structures call for locking and/or timestamping mechanisms, and may induce I/O to write changes to disk. Locking may lead to lock-contention, while both timestamping and deletion markers induce per-tuple storage and run-time overheads, where tuple "aliveness" needs to be verified during every scan. Besides, with update information being distributed over distinct data structures, conflict detection/resolution and recovery can become complex to manage. By capturing transaction updates (i.e. the write-set) in private, memory- resident trans-PDTs, which are only written to the WAL during commit, and combining that with multiversion concurrency control, we avoid above issues, and allow read queries to always operate on the most recent snapshot, transparently, at a negligible cost.

Finally, a stacked PDT approach also allows for a transparent and fully concurrent implementation of checkpointing operations (also referred to as *tuple mover* or *mergeout* in the literature). By locking down the read-PDT, but still allowing concurrent transactions to commit into the write-PDT, we can safely rebuild a table in the background, while concurrent updates gathered in the write-PDT remain compatible with the new table image after the checkpoint completes. Only post-checkpoint index maintenance requires some additional work, for which we provide efficient solutions that only need to analyze updates contained in concurrently built PDTs. SQL Server employs multiple write-store deltas, the oldest of which are locked down for tuple movement, while newer ones can be inserted into concurrently. However, deletion (and therefore also modify) is not supported during tuple movement, as the global bit map is locked down. In Vertica, tuple movement relies on relatively complex locking and timestamping mechanisms, which also prohibit delete and modify while a "tuple mover" lock is being held. Both SQL Server and Vertica do allow partial rebuilding of tables, while Vectorwise almost always performs a linear (i.e. scan-based) table rebuild. During this process, pages without updates are re-linked on disk, to avoid needless reads and writes during this process. We did not evaluate table checkpointing in much depth, as it depends heavily on the I/O configuration and workload characteristics. In this area, we do, however, benefit from the fact that we designed both compression and decompression to be highly CPU efficient. Recent work in the area of compressed table rebuilds can be found in the context of SAP Hyrise in [KKG+11].

Until a checkpoint operation is run, query evaluation performance of all

column-stores with differential updates will degrade over time (assuming updates keep coming in). For Vectorwise, this mostly manifests itself as increased memory pressure from PDT deltas, combined with somewhat slower scans due to higher update ratios. In case of disk-resident, row-oriented deltas, memory pressure is less of an issue, but deltas need to be read from disk during a scan, inducing additional I/O, and, in case of ordered tables, be merged in using (slow) value-based techniques. Therefore, over time, this approach will degrade towards traditional row-store scan performance, where all attributes are always scanned from disk. In our experiments involving value-based merging, we did not even factor such full tuple retrieval costs into account (those depend heavily on tuple width), focusing only on the I/O penalty due to retrieval of the sort key columns that are needed to perform value-based merging of deltas that are considered to be memory resident. In this context, it is also worth considering the dangers of implementing modify as a delete followed by insert. In the worst case, when modifying all values in a single column, the "hybrid" architecture would result in the entire table being rewritten into row-store format! While, in Vectorwise, the Modify executes using the scan of only the modified column, plus the memory cost of storing the new values. In such a scenario, Vectorwise could benefit from per-column checkpointing techniques, which is still an area of future work.

Although automatic checkpoints will free up PDT memory regularly, during times of high system load it may be beneficial to delay such a checkpoint. As memory is finite, it can not be delayed indefinitely, however. Because a checkpoint has a relatively long running time, we plan to investigate the addition of one or more PDT layers on solid state disk (SSD), which sits in between main memory and hard disk storage in terms of access bandwidth, latency, and storage capacity. Such a PDT layer could enable fast propagation of memory resident (read-)PDT updates to SSD, without performing a checkpoint operation, but with the benefit of freeing up the bulk of PDT memory. We could either add a single PDT layer below the read-PDT, subject to the same copy-on-write and update propagation mechanisms as the global write- and read-PDTs, or simply flush a read-PDT into an immutable "run" of updates on SSD (allowing multiple runs to be generated), in combination with an N-way merging process during MergeScan, akin to the ideas presented in Vertica and MaSM [ACA+11] (which focuses on SSDs), but rather positionally.

We also believe that PDT updates may be interesting in the context of row-oriented OLTP systems. Although in a row-store system we do not get the I/O advantages associated with positional merging in a column-store, we can still get the CPU benefits. More importantly, however, PDTs could be used to implement a lock-free transactional layer, which could be combined with *partial checkpointing* techniques, where only those regions of (committed) read-PDT updates are checkpointed that exceed a certain per-page update ratio. Such fine-grained checkpointing is something that suits a row-wise slotted page layout much better than a columnar storage layout. This way, PDTs may be used as a light-weight transactional layer, with the aim of avoiding the dangers associated with "heavy-weight" in-place updates, against which Jim Gray already warned long ago [Gra81].

Another area where differential updates using PDTs may be of interest is within the *Hadoop distributed file system* (HDFS) [Apaa], due to its *append-only* nature (i.e. no in-place update support). HDFS supports multiple underlying

file formats, which may contain objects, or rows, with multivalued (i.e. "complex"), attributes, represented as an arbitrary size array or a map (i.e. two aligned key-value columns). One of the more recent formats is the *optimized record columnar file* (ORC File), employed by Apache Hive [Apab, HCG+14], a SQL-like data warehousing solution on top of Hadoop. In ORC file, object attribute data is stored in a compressed columnar format. For multivalued attributes, however, the attribute column contains pointers to a (dense) tuple range in an *additional* column, containing the multi-valued attribute data. Such a layout resembles the clustering table layout that is available in Vectorwise through the use of join indices. Apache Hive developers are currently undertaking steps to add value-based differential update and transaction support. We believe that the positional update techniques outlined in this thesis may provide a more efficient alternative, but research in this area is still needed. Steps in this direction are being taken by Actian's *Vector in Hadoop*[1], also known as "Project Vortex", which aims to extend Vectorwise's high-performance relational engine to a distributed HDFS setting.

Finally, a couple of recent works borrow from our ideas around PDTs. In [MG12], a *count index* is described, which can be used to efficiently insert, delete and modify run-length encoded sequences, using techniques that are similar to what we use to update our RLE compressed join indices. Furthermore, according to [FKN12], the HyPer system [KN11] also makes use of a PDT-like structure, but restricts it to organize offsets (and ranges) of *deleted* tuples only. HyPer belongs to a new class of in-memory hybrid OLTP and OLAP database systems, to which SAP HANA [FML+12] also belongs. It uses hardware assisted "page shadowing" techniques to implement ACID properties using snapshot isolation, where the virtual memory manager is used to maintain consistent snapshot copies of a database. The concept of "page temperature" is introduced to classify pages based on update frequency, using hot, cooling, cold and frozen labels. Updates can only go to hot and cooling pages, while frozen pages are compressed to save memory and speed-up analytic workloads. Although HyPer's architecture differs heavily from traditional RDBMS designs, this shows some resemblance to a distinction between a read-optimized and a write-optimized component, where updates to read-optimized pages are never performed in-place, but rather buffered and delayed to amortize their high cost. It is encouraging to see PDTs being used (for tuple deletion) in the context of this highly innovative and fast main memory DBMS.

[1] Vectorwise has been recently renamed to Vector

Appendix A

Information Retrieval using Vectorwise

A.1 Introduction

Requirements of database management (DB) and information retrieval (IR) systems overlap more and more. Database systems are being applied to scenarios where features such as text search and similarity scoring on multiple attributes become crucial. Many information retrieval systems are being extended beyond plain text, to rank semi-structured documents marked up in XML, or maintain ontologies or thesauri. In both areas, these new features are usually implemented using specialized solutions limited in their features and performance.

Full integration of DB and IR has been considered highly desirable, see e.g. [CRW05, AY05]. Yet, none of the attempts into this direction has been very successful. The explanation can be sought in what has been termed the 'structure chasm' [HED+03]: database research builds upon the idea that all data should satisfy a pre-defined schema, and the natural language text documents of concern to information retrieval do not match this database application scenario. Still, the structure chasm does not explain why IR systems do not use database technology to alleviate their data management tasks during index construction and document ranking. In practice however, custom-built information retrieval engines have always outperformed generic database technology, especially when also taking into account the trade-off between run-time performance and resources needed.

A.1.1 Contributions

The aim of this chapter is twofold. First, it demonstrates the advantage, in terms of *flexibility*, of using standard relational algebra to formulate IR retrieval models. Secondly, it shows that, by employing a *hardware-conscious* DBMS architecture, it is possible to achieve performance, both in terms of *efficiency* and *effectiveness*, that can compete with leading, customized IR systems. To prove this, we use Vectorwise to participate in the 2006 TREC Terabyte Track (TREC-TB) [BCS06], a terabyte-scale information retrieval benchmarking task, obtaining competitive results. Running TREC-TB on top of a DBMS efficiently,

is something that has never been demonstrated to be realizable before, and could therefore be seen as a step towards closing the gap between DBMS and IR systems.

A.1.2 Outline

In Section A.2 we provide an overview of the TREC-TB benchmark and the table layout we use to index its document collection. In Section A.3 we describe a relational approach to keyword search, together with several commonly applied IR optimizations and benchmark results. To evaluate the performance of our system with respect to custom built IR systems, we provide results of our 2006 TREC-TB submission in Section A.4. We discuss related work in Section A.5, before concluding in Section A.6

A.2 TREC-TB Setup

A.2.1 Overview

From 2004 to 2006, the Text REtrieval Conference (TREC) organized a "TeraByte Track" [CSS05] as a large-scale text retrieval testbed. The TeraByte Track (TREC-TB) consists of the GOV2 document collection, together with ad-hoc retrieval tasks to evaluate system performance in terms of both effectiveness and efficiency. The data set consists of 25 million web documents, crawled from the *.gov* domain, with a total size of 426GB. System efficiency is measured by total execution time of 50,000 keyword-search queries. Effectiveness is evaluated by early precision (*p@20*) on a subset of 50 preselected queries for which relevance judgments are available. Table A.1 shows the performance of the leading systems on the 2005 TREC-TB efficiency task.

Run	**p@20**	**CPUs**	**Time per query (ms)**
MU05TBy3	0.5550	8	24
uwmtEwteD10	0.3900	2	27
MU05TBy1	0.5620	8	42
zetdist	0.5300	8	58
pisaEff4	0.3420	23	143

Table A.1: Top results for TREC-TB 2005

We ran the 2005 TREC-TB on top of Vectorwise, using a single 3GHz Pentium Xeon CPU, 4GB of RAM, and a software RAID system consisting of 12 disks. For the distributed experiments in Section A.3.3, we used a LAN of 8 machines, equipped with dual-core, 2GHz Athlon64X2 CPUs and 2GB RAM.

A.2.2 Indexing

Indexing the TREC-TB document collection entails three main phases: parsing, constructing an inverted index structure using relational tables, and index compression. Parsing is done using an external program, that scans the collection and filters out markup and stop words. For the remaining text, the

parser returns (docid, term) pairs (DT) for each term it encounters, with terms being stemmed using a Porter stemmer [Por80], converted to lowercase, and scrambled into 64bit integers. Our scrambling function produces a one-to-one mapping from its input string to its integer representation, as long as no other input string has the same thirteen character long prefix. This is achieved by iterating over the input string, and on each iteration multiplying the current integer result by twenty-seven and adding the i'th characters offset from the character 'a' + 1. Furthermore, the parser generates a unique *docid* identifier for each document encountered, and outputs it, together with the documents name and length (in number of terms).

To index the data, we used an *inverted list* data-structure (see Section 3.3.3), represented by a relational table. To build this index from the DT output of the parser, the following relational query, which sorts and then aggregates on (term, docid) pairs, was used:

```
# TD computation using DT
  Aggr(
    Sort(
      Scan(DT, [docid, term] ),
      [ term, docid ] ),
    [ term, docid ],
    [ tf = count() ] )
```

The *[term,docid,tf]* (TD) table, holds for each term, the IDs of the documents the term appears in (*docid*), and the number of times the term occurs within a given document (*tf*). The table is ordered on *(term,docid)*, which allows the *term* column to be replaced by a range index onto *[docid,tf]*, and allows the occurrence lists of two arbitrary terms to be combined efficiently using merge-join. Additionally, per-document information is kept in a separate *[docid, name, length]* document table D, and per term information *[term, ftd]* in table T. The relational table layout, together with the amount of storage each field occupies, is summarized in table A.2.

Compression

As Table A.2 shows, the full index (the D, T and TD tables), occupies approximately 29 GB uncompressed when we ignore the *term* column in TD, and replace it with a range index of negligible size. We applied Vectorwise's PFOR and PFOR-DELTA light-weight column compression algorithms from Section 5.3 to reduce the total size of this index to roughly 9GB. The benefit of this is twofold. First of all, due to the minimal decompression overhead of the compression algorithms, I/O bandwidth utilization is improved, as the data gets decompressed only when it is used, upon crossing the RAM-CPU Cache boundary. Second, the compressed index requires less memory to make it fully main-memory resident. Even if it does not fit fully, more data can be cached in RAM in compressed form, improving overall performance.

A.3 Querying

A.3.1 Keyword Search Using Relational Algebra

Keyword search in a DBMS boils down to retrieving all the documents in which some or all of the query terms occur. Such a boolean retrieval approach can be

symbol	column name	semantic	sorted type		compression scheme	bits
DT – 12.3 Gtuples, output of parsing						
D	docid	document id	int	Y	none	32
T	term	term code	long	N	none	64
TD – 3.5 Gtuples, document-level index						
T	term	term code	long	Y	$\text{PFD}_{b=1}$	2.13
D	docid	document id	int	Y	$\text{PFD}_{b=8}$	11.98
$f_{D,T}$	tf	frequency of T in D	int	N	$\text{PF}_{b=5}$	5.91
$\omega_{D,T}$	score	score of T in D	float	N	none	32
$\omega'_{D,T}$	scoreQ	quantized score	int	N	$\text{PF}_{b=8}$	8.00
D – 25 Mtuples, output of parsing, per-document information						
D	docid	document id	int	Y	none	32
	docname	document name	str	Y	none	88
$\|D\|$	doclen	document length	int	N	none	32
T – 12 Mtuples, per-term information						
T	term	term code	long	Y	none	64
$f_{T,D}$	ftd	#documents with T	int	N	none	32
Global constants						
k_1	k1	BM25 parameter (0.8)				
b	b	BM25 parameter (0.3)				
f_D	numdocs	number of documents ($25M$)				
$avgdl$	avgdoclen	average document length (491)				
compression: **PF**=PFOR, **PFD**=PFOR-DELTA, all with base=0						

Table A.2: Database tables and constants used

formulated in relational algebra as a series of join operations over inverted lists, with boolean AND and OR mapping to *Join* and *OuterJoin* respectively. For example, a query *"information AND (storing OR retrieval)"* can be translated to:

```
MergeJoin(
  ScanSelect( TD1=TD, TD1.term="information" ),
  MergeOuterJoin(
    ScanSelect( TD2=TD, TD2.term="storing" ),
    ScanSelect( TD3=TD, TD3.term="retrieval" )))
```

As the results for the runs **BoolAND** and **BoolOR** from Table A.3 show, simple boolean queries without ranking result in very low precision. To address the low effectiveness, we present results with the Okapi BM25 [RWB98] retrieval model. The document score of a given query is expressed as:

$$S^{(D)}_{BM25} = \sum_{i=1}^{|Q|} \omega_{D,T_i} \tag{A.1}$$

$$\omega_{D,T} = \log(\frac{f_D}{f_{T,D}}) \cdot \frac{(k_1+1)\cdot f_{D,T}}{f_{D,T} + k_1 \cdot ((1-b) + b \cdot \frac{|D|}{avgdl})} \tag{A.2}$$

Given a query with $|Q|$ terms, the score of each document $S^{(D)}_{BM25}$ is a sum of scores of each query term for this document $\omega_{D,T}$. The per-term document scores $\omega_{D,T}$ are computed as a function of the total number of documents (f_D),

the number of documents containing term T ($f_{T,D}$), the frequency of T within D ($f_{D,T}$), the document length ($|D|$), and the average document length ($avgdl$) over the whole collection. Variables k_1 and b represent two predefined constants. A relational query that finds the top 20 documents using this formula for a 2-term query could look like:

```
TopN(
  Project(
    MergeJoin(
      MergeOuterJoin(
        ScanSelect( TD1=TD, TD1.term=t1_term ),
        ScanSelect( TD2=TD, TD2.term=t2_term ),
        TD1.docid=TD2.docid),
      Scan( D ),
      D.docid=MAX(TD1.docid,TD2.docid))
    [ D.docname, score=BM25(TD1.tf,D.doclen,t1_ftd)
      +BM25(TD2.tf,D.doclen,t2_ftd) ]),
  [ score DESC ], 20)
```

where `score=BM25(TD2.tf,D.doclen,t2_ftd)` is used as a shorthand, the real query contains the full BM25 formula.

Running above BM25 query requires a significant amount of processing time, as run **BM25** in Table A.3 illustrates. This is not strange, however, considering that the average length of the 50.000 TREC-TB queries is 2.3 terms, with each term occurring in 775 thousand documents on average. There is, however, still room for improvement in terms of average query execution time.

A.3.2 Performance Optimizations

Run name (+ added feature)	p@20	Avg.query time (ms), cold data	Avg.query time (ms), hot data
BoolAND	0.0130	76	12
BoolOR	0.0000	133	80
BM25	0.5460	440	342
BM25T (+Two-pass)	0.5470	198	72
BM25TC (+Compression)	0.5470	158	73
BM25TCM (+Materialization)	0.5470	155	29
BM25TCMQ8 (+Quant.8-bit)	0.5490	118	28

Table A.3: Vectorwise TREC-TB Experiments

In this section we show a set of representative IR optimization techniques that allowed Vectorwise to be competitive with the leading TREC participants.

First of all, The BM25 retrieval model scores each document, regardless the number of matching query terms. Given the observation that we are only interested in the top-N most relevant documents, we can refrain from computing the score for documents that are highly unlikely to make it into the top-N. Relying on a heuristic that documents that contain more query terms are likely to obtain a better score [BCH$^+$03], we can obtain a significant performance improvement by following a **two-pass** strategy. In the first pass, we retrieve only the documents that contain *all* query terms, using a conventional *MergeJoin* instead of a *MergeOuterJoin*. Only if the first pass does not return enough

results, we execute a second pass using the less restrictive *MergeOuterJoin*. Run **BM25T** in table A.3 illustrates the performance gain, where roughly 15% of the 50.000 queries required a second pass.

Second, a large part of the cost of processing inverted lists is related to reading these from disk. Most IR systems use data compression to reduce this time. Using Vectorwise's built in **compression**, we were able to reduce the sizes of the *docid* and tf columns, which constitute the major part of total I/O, from 32 to 11.98 and 5.98 bits per tuple, respectively. To compress the partially ordered *docid* column, we used PFOR-DELTA compression with a code word size of 8 bits. For the small integer tf values, we used PFOR, with a 5-bit code word size. The **BM25TC** run in Table A.3 shows that this significantly improves the cold run, where all data needs to be read from disk. The time of the hot run did not change significantly, thanks to the high performance of the decompression routines implemented in Vectorwise.

The BM25 retrieval model aggregates the $\omega_{D,T}$ scores, which are query independent (see Eq. A.1 and A.2). Once its tuning parameters k_1 and b have been fixed, $\omega_{D,T}$ values may be precomputed, and can be stored as a separate *score* column in the TD table. This **score materialization** not only saves the cost of computing the per-term document score, it additionally allows us to perform a join with the document table only for the top-n documents, since the per-document properties are not needed for score computation anymore, only to retrieve the names of the top-ranked documents. As the results of the **BM25TCM** run show, this reduces the in-memory processing time significantly. However, the *cold* run did not improve, since the I/O overhead increased, as we now read 32-bit floating point $\omega_{D,T}$ values instead of compressed 5.98-bit term frequencies $f_{D,T}$. We were able to **quantize** the range of floating point scores into 8-bit integer *score ranks* [AM05a], without loss of precision, using the following *linear Global-By-Value* quantization

$$\omega'_{D,T} = \left\lfloor q \cdot \frac{\omega_{D,T} - L}{U - L + \epsilon} \right\rfloor + 1,$$

where L and U are the minimum and maximum values of $\omega_{D,T}$ in the entire collection. This produces integer values between 1 and q. We used a value of $q = 256$, significantly reducing the data size, and therefore the amount of I/O, which resulted in the performance results labeled **BM25TCMQ8**.

A.3.3 Distributed Execution

Text retrieval lends itself well for distributed execution, as we can easily split up the document collection into N partitions, and let each partition be indexed by its own server node. An incoming query can then be broadcast to all indexing nodes, with each of them returning its local top-N documents for that query. These per-node results can then be merged into a global top-N to produce the final result.

To investigate the scalability of our system, we ran distributed experiments on our LAN, using 8 partitions, each indexed by a separate machine. Queries are submitted to a *broker* program, which broadcasts each query to all eight nodes, and merges the local document rankings returned by each node into a global ranking. Thanks to Vectorwise's data compression, the whole index (9GB), could be kept in RAM, so that I/O is eliminated as a performance factor.

	Average query time (ms)		Average per-query server response time (ms)		
	absolute	amortized	min	average	max
Full TREC-TB run (hot data)					
Sequential	23.1				
8 servers	11.26		5.50	6.39	11.00
Using less servers (1 stream, fixed partition size)					
4 servers	9.21		5.92	6.78	9.06
2 servers	7.30		6.46	6.83	7.20
1 server	7.41		7.34	7.34	7.34
Increasing the concurrency (8 servers)					
1 stream	11.24	11.26	5.50	6.39	11.00
2 streams	9.61	4.86	5.56	6.92	9.36
4 streams	14.30	3.64	5.81	8.56	13.99
8 streams	25.46	3.26	6.21	12.28	25.07

Table A.4: Performance of the distributed runs

However, as the results in A.4 show, the *speedup* using 8 machines is far from perfect (it decreases from 23.1 only to 11.26 msec). The main cause for the non-linear speedup is load imbalance, as is shown in the right half of Table A.4: with increased number of database servers, the average per-server running times start to vary significantly. With 8 servers, the slowest one (which determines the overall query latency) takes twice as long as the fastest (11 vs. 5.5 msec). In a real IR system, however, such load imbalance affects latency but not throughput, as the system will be handling multiple queries continuously and differences even out. This is currently modeled in the TREC terabyte efficiency track by submitting a limited number of concurrent query "streams" to the system. The lower part of Table A.4 shows that with an increased amount of concurrency, latency deteriorates much less than linear (i.e. throughput improves). As a result, throughput does scale linearly, as 8 servers are able to process more than 300 queries per second, taking an amortized 3.26 msec per query only (vs. 23 msec for one server).

A.4 TREC-TB Results

Our TREC-TB setup was used to participate in the last official Terabyte Track [BCS06] in 2006. For this benchmark, only the set of topics, i.e. the queries, changed. The document collection, and our mapping of it to relational tables, remained the same. Besides our distributed setup, as described in Section A.3.3, we also submitted runs a single node "hot" run, i.e. where the entire index was preloaded into the main memory of a single machine. For this, we used a relatively old SMP machine (even at that time), containing four 1.4GHz AMD Opteron CPUs and 16GB of RAM. The official results from [BCS06] are replicated in Table A.5, where the two rows starting with CWI06, represent Vectorwise's best results in terms of query latency and precision, respectively.

Vectorwise is the winner in terms of query latency and throughput. However,

Run	Number of CPUs	System Cost (USD)	Query Streams	Latency			Throughput			Effectiveness	
				Measured (ms/query)	CPU-normalized	cost-normalized	measured (queries/s)	CPU-normalized	cost-normalized	P@20 (801-850)	MRR (901-1081)
CWI06DIST8	16	6,400	4	13	211	85	185.5	11.6	29.0	0.4680	0.181
CWI06MEM4	4	10,000	4	80	322	805	48.7	12.2	4.9	0.4720	0.190
humTE06i3	1	5,000	1	1,680	1,680	8,400	0.6	0.6	0.1	0.3690	0.123
humTE06v2	1	5,000	1	4,630	4,630	23,150	0.2	0.2	0.0	0.4290	0.373
mpiiotopk2p	2	5,000	4	29	57	143	35.0	17.5	7.0	0.4330	0.280
mpiiotopkpar	2	5,000	4	74	148	369	13.6	6.8	2.7	0.4280	0.291
MU06TBy6	1	500	4	55	55	28	18.2	18.2	36.4	0.4890	0.271
MU06TBy2	1	500	1	229	229	114	4.4	4.4	8.8	0.5050	0.256
p6tbep8	1	1,400	1	109	109	153	9.1	9.1	6.5	0.3890	0.254
p6tbeb	1	1,400	1	167	167	234	6.0	6.0	4.3	0.4540	0.244
rmit06effic	2	4,000	1	2,202	4,404	8,808	0.5	0.2	0.1	0.4650	0.258
THUTeraEff02	4	2,000	1	534	2,136	1,068	1.9	0.5	0.9	0.1500	0.222
THUTeraEff01	4	2,000	1	808	3.232	1,616	1.2	0.3	0.6	0.3920	0.246
uwmtFdcp03	1	1,800	1	13	13	23	80.0	80.0	44.4	0.4110	0.164
uwmtFdcp12	1	1,800	1	32	32	58	31.4	31.4	17.4	0.4790	0.219

Table A.5: 2006 TREC-TB efficiency results, showing fastest run (according to avg. query latency) and best run (according to precision, P@20) for each participating group. Source: [BCS06]

our submission also has the highest CPU count and system cost, which renders comparison of absolute numbers unfair. In an effort to address this, TREC computed latency and throughput numbers normalized to both the number of CPUs and system cost. Although not perfect, this does allow for fairer comparisons.

Looking at the efficiency results (i.e. the top row of each group) CPU-normalized latency is where our CWI06DIST8 run performs worst, achieving a fifth place overall. Our score of 211 ms/query is still significantly better than the 1208ms average of the remaining seven systems though. For CPU-normalized throughput, we do somewhat better, scoring fourth. However, now our 11.6 queries/s is worse than the average of 18.0. Looking at the cost-normalized numbers, CWI06DIST8 scores a third spot, both for latency and throughput. We also beat the average in both cases: 85 versus 2660 for latency, and 29 versus 13.6 for throughput. An interesting observation is that most systems that beat us consistently on the normalized scores, i.e. mpiiotopk2p, MU06TBy6 and uwmtFdep03, perform considerably worse in terms of precision (P@20). The exception to this is MU06TBy6, with very solid scores overall.

Looking at the effectiveness scores, we see *precision* (P@20), which represents the fraction of top-20 documents that is considered relevant, and *mean reciprocal rank* (MRR), where the reciprocal rank of a single query response is defined as the inverse of the rank ($1/rank$) of the first correct answer, with correctness judged by a jury. For example, an MRR of 0.2 means that, on average, the first correct answer is in fifth spot in the ranking. Note that the two CWI runs

score very similar, even though we used the same BM25 ranking formula in both scenarios. The difference can be explained by the fact that in the distributed runs each node submits a *local* top-20 to the broker, which then merges them into a global top-20. In case of ties in the score, the ordering of results can vary arbitrarily. Because the effects are very small, and the CWI06DIST8 run beats the CWI06MEM4 run in all other respects, the latter can safely be ignored. Most other groups submitted different configurations for both the efficiency and effectiveness runs, in general sacrificing one in favor of the other.

Comparing our effectiveness scores with the second rows of the other groups, we see that Vectorwise scores very decently in terms of precision, achieving a third place. In terms of *mean reciprocal rank* (MRR) we score worse than most other systems. We did not have the means to investigate this further, but we suspect it to be caused by relative imprecision of our quantized scores (8-bit integers) and/or their potential for unresolvable collisions.

A.5 Related work

Integration of DB and IR processing has recently been discussed in [CRW05]. The authors present a set of motivating examples, discuss different approaches of combining area-specific processing techniques, and propose an extension to the relational algebra that provides various IR features, e.g. a top-k operator. They also discuss the new challenges this algebra brings for relational query optimizers.

A good summary of the DB-community view on integration with IR technology was presented during the recent SIGMOD panel [AY05]. While most researchers discuss DB and IR integration within a DBMS, our approach is to rather provide IR applications with features necessary for the efficient execution of their tasks. In this sense it is similar to [GFHR97], where the authors store inverted lists in a Microsoft SQL Server and use SQL queries for keyword search. Similarly, in [GBS04] the data is distributed over a PC cluster, and an analysis of the impact of concurrent updates is provided. Our approach extends this previous work, by showing how a much wider series of IR optimization techniques can be translated to database queries. These techniques and their effectiveness/performance trade-off are further demonstrated on a much larger collection (500GB TeraByte TREC vs. 500MB in [GBS04]) and show significantly faster retrieval performance.

In the TREC benchmark there were a few attempts to use database technology, e.g. [MKD99]. However, most of these systems used a DBMS for effectiveness tasks only, where the system efficiency was not an issue. Only one TeraByte TREC submission used a system built on top of the MySQL DBMS [CCS04], but its precision and speed (5 sec per query) were disappointing compared to other participants.

There still is a large group of IR efficiency optimization techniques not discussed in this paper. For example, Buckley [BL85] presented an optimization technique in which, during *term-at-a-time* top-r search, execution stops when the score difference of the r-th and $r+1$-th document is larger than the sum of the maximum scores of the remaining attributes. Another pruning approach is the well-known Fagin algorithm [FLN01], recently extended with probabilistic pruning [TWS04]. The final interesting group of optimizations exploit word

positions for improved retrieval effectiveness (e.g. [MC05]). We believe all these methods can be implemented on top of a DBMS using techniques similar to the ones presented in this paper.

A.6 Conclusion

In this chapter we have shown that it is possible to run terabyte-scale information retrieval tasks on top of a relational database engine. Furthermore, we have shown that we can easily implement several standard IR optimization techniques, and that these techniques allow us to rival customized IR systems in terms of performance. This work presents a step towards the integration of DB and IR systems, with some of the key ingredients needed to achieve this result being: Vectorwise's raw speed, light-weight data compression, and distributed execution.

Considering recent advances in hardware, especially increased memory sizes and bandwidth, together with higher levels of parallelism in multi-core CPUs, one could envision running the distributed experiments – i.e. using a partitioned document collection – on a single machine. This would eliminate network latency, and significantly reduce system costs. However, scalability is at risk, with memory bandwidth as the likely bottleneck. It would be interesting to see how far a NUMA architecture could be pushed on such a workload.

Bibliography

[ACA+11] Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. MaSM: efficient online updates in data warehouses. In *Proc. SIGMOD*, pages 865–876, 2011.

[Act] Actian. *Actian Analytics Database - Vector 3.5*. `http://bigdata.actian.com/vector-3.5`.

[ADH02] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[ADHS01] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, 2001.

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, 1999.

[Adv] Advanced Micro Devices Inc. *AMD Unleashes First-Ever 5 GHz Processor*. `http://www.amd.com/us/press-releases/Pages/amd-unleashes-2013jun11.aspx`.

[AHKB00] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. *ACM SIGARCH Computer Architecture News*, 28(2):248–259, 2000.

[AM05a] V. N. Anh and A. Moffat. Simplified Similarity Scoring Using Term Ranks. In *Proc. SIGIR*, 2005.

[AM05b] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[AMF06] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. SIGMOD*, 2006.

[AMH08] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[Ani10] Kamil Anikiej. Multi-core parallelization of vectorized query execution. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, July 2010.

[Apaa] Apache. *Hadoop*. `http://hadoop.apache.org`.

[Apab] Apache. *Hive*. `http://hive.apache.org`.

[Apac] Apache. *Lucene*. `https://lucene.apache.org/`.

[Ass95] Assar, Mahmud, Nemazie, Siamack, Estakhri, Petro. Flash memory mass storage architecture, February 1995.

[AY05] S. Amer-Yahia. Report on the DB/IR Panel at Sigmod 2005. *SIGMOD Record*, 34(4), 2005.

[Bac78] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD*, 1995.

[BC07] Stefan Büttcher and Charles LA Clarke. Index compression is good, especially for random access. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 761–770. ACM, 2007.

[BCH+03] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*, 2003.

[BCS06] S. Büttcher, C. L. A. Clarke, and I. Soboroff. The TREC 2006 Terabyte Track. In *Proc. TREC*, 2006.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 283–296. ACM, 2009.

[BK99] P. Boncz and M. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal*, 8(2):101–119, 1999.

[BL85] C. Buckley and A. F. Lewit. Optimization of inverted list search. In *Proc. SIGIR*, 1985.

[BMK99] Peter A Boncz, Stefan Manegold, and Martin L Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[Bon02] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, May 2002.

[BS13] Nathan Beckmann and Daniel Sanchez. Jigsaw: scalable software-defined caches. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 213–224. IEEE Press, 2013.

[BW94] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center, May 1994.

[BYRN+99] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[BZN05] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, 2005.

[CAGM04] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proc. ICDE*, 2004.

[CB74] Donald D Chamberlin and Raymond F Boyce. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[CBB+13] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, et al. Unicorn: A system for searching the social graph. *Proceedings of the VLDB Endowment*, 6(11):1150–1161, 2013.

[CCS04] C. L. A. Clarke, N. Craswell, and I. Soboroff. Overview of the TREC 2004 Terabyte Track. In *Proc. TREC*, 2004.

[CD85] H.-T. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proc. VLDB*, 1985.

[CGF07] G. Canahuate, M. Gibas, and H. Ferhatosmanoglu. Update conscious bitmap indices. In *Proc. SSDBM*, 2007.

[CGK01] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. *SIGMOD Rec.*, 30(2):271–282, 2001.

[CGM01] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *Proc. SIGMOD*, 2001.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[CHZ+08] Roberto Cornacchia, Sándor Héman, Marcin Zukowski, Arjen P de Vries, and Peter Boncz. Flexible and efficient IR using array databases. *The VLDB Journal*, 17(1):151–168, 2008.

[CK85] A. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, 1985.

[CLG+94] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod79] Edgar F Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434, 1979.

[Col13] Robert Colwell. The Chip Design Game at the End of Moore's Law. In *Keynote presentation at Hot Chips conference*, 2013.

[Coo01] C. Cook. *Database Architecture: The Storage Engine*, 2001. `http://msdn.microsoft.com/library`.

[COO08] Xuedong Chen, Partick O'Neil, and Elisabeth O'Neil. Adjoined dimension column clustering to improve data warehouse query performance. In *Proc. ICDE*, 2008.

[CR93] C.-M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proc. VLDB*, 1993.

[CRF08] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proc. SIGMOD*, 2008.

[CRW05] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In *Proc. CIDR*, 2005.

[CSS05] C. L. A. Clarke, F. Scholer, and I. Soboroff. The TREC 2005 Terabyte Track. In *Proc. TREC*, 2005.

[Den68] Peter J Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, 1968.

[DGR+74] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[Exa] Exasol. `http://www.exasol.com`.

[FDN+01] David J Frank, Robert H Dennard, Edward Nowak, Paul M Solomon, Yuan Taur, and Hon-Sum Philip Wong. Device scaling limits of Si MOSFETs and their application dependencies. *Proceedings of the IEEE*, 89(3):259–288, 2001.

[Fer94] Phillip M Fernandez. Red brick warehouse: a read-mostly RDBMS for open SMP platforms. In *ACM SIGMOD Record*, volume 23, page 492. ACM, 1994.

[FKN12] Florian Funke, Alfons Kemper, and Thomas Neumann. Compacting transactional data in hybrid OLTP&OLAP databases. *Proceedings of the VLDB Endowment*, 5(11):1424–1435, 2012.

[FLN01] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. PODS*, pages 102–113, Santa Barbara, CA, USA, 2001.

[Fly72] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21(9):948–960, 1972.

[FML+12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[FMM+12] Martin Fuechsle, Jill A Miwa, Suddhasatta Mahapatra, Hoon Ryu, Sunhee Lee, Oliver Warschkow, Lloyd CL Hollenberg, Gerhard Klimeck, and Michelle Y Simmons. A single-atom transistor. *Nature Nanotechnology*, 7(4):242–246, 2012.

[GBS04] Torsten Grabs, Klemens Bhoem, and Hans-Jorg Schek. PowerDB-IR: scalable information retrieval and storage with a cluster of databases. *Knowledge and Information Systems*, 6(4):465–505, 2004.

[GFHR97] D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating structured data and text: A relational approach. *JASIS*, 48(2), 1997.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *Proc. VLDB*, pages 144–154, 1981.

[Gra90] Goetz Graefe. *Encapsulation of parallelism in the Volcano query processing system*, volume 19. ACM, 1990.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[Gra03] Goetz Graefe. Sorting and indexing with partitioned b-trees. In *Proc. CIDR*, 2003.

[Gra07] Goetz Graefe. Efficient Columnar Storage in B-trees. *SIGMOD Record*, 36(1), 2007.

[Gre] Greenplum Database. `http://www.greenplum.com`.

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proc. ICDE*, 1998.

[GS91] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, 1991.

[HBZN10] S.A.B.C. Heman, P.A. Boncz, M. Zukowski, and N.J. Nes. Column-store database architecture utilizing positional delta tree update system and methods, September 16 2010. US Patent App. 12/719,825.

[HBZN13a] S.A.B.C. Heman, P.A. Boncz, M. Zukowski, and N.J. Nes. High-performance database engine implementing a positional delta tree update system, July 11 2013. US Patent App. 13/417,205.

[HBZN13b] S.A.B.C. Heman, P.A. Boncz, M. Zukowski, and N.J. Nes. Methods of operating a column-store database engine utilizing a positional delta tree update system, July 11 2013. US Patent App. 13/417,204.

[HCG+14] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major technical advancements in apache hive. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1235–1246. ACM, 2014.

[HED+03] Alon Y. Halevy, Oren Etzioni, AnHai Doan, Zachary G. Ives, Jayant Madhavan, Luke McDowell, and Igor Tatarinov. Crossing the structure chasm. In *Proc. CIDR*, 2003.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *Proc. VLDB*, 2006.

[HNZB07] Sándor Héman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd international workshop on Data management on new hardware*, page 4. ACM, 2007.

[HP11] J. Hennessy and D. Patterson. *Computer Organization and Design: The Hardware/Software interface (fifth edition)*. Morgan Kaufmann Publishers Inc., 2011.

[HP12] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach (fifth edition)*. Morgan Kaufmann Publishers Inc., 2012.

[HRSD07] Allison L Holloway, Vijayshankar Raman, Garret Swart, and David J DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 389–400. ACM, 2007.

[HSA05] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *Proc. SIGMOD*, 2005.

[Huf52] D.A. Huffman. A method for construction of minimum redundancy codes. volume 40, pages 1098–1101, 1952.

[HZdVB06] Sándor Héman, Marcin Zukowski, Arjen P de Vries, and Peter A Boncz. MonetDB/X100 at the 2006 TREC Terabyte Track. In *TREC*. Citeseer, 2006.

[HZN+10] Sándor Héman, Marcin Zukowski, Niels J Nes, Lefteris Sidirourgos, and Peter Boncz. Positional update handling in column stores. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 543–554. ACM, 2010.

[IBM07] IBM Corp. *Cell Broadband Engine*, 2007.

[Int] Intel Corp. *4th Generation Intel® Core™ i7 Processor*. `http://ark.intel.com/products/76642/Intel-Core-i7-4770R-Processor-6M-Cache-up-to-3_90-GHz`.

[Int09] Intel Corp. *An Introduction to the Intel® QuickPath Interconnect*, January 2009.

[Int10] Intel Corp. *Intel® Itanium® Architecture Software Developers Manual, Rev. 2.3*, May 2010.

[Int12a] Intel Corp. *Intel® Architecture Instruction Set Extensions Programming Reference*, 2012. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[Int12b] Intel Corp. *Intel® Xeon Phi™ Product Family*, February 2012. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[Int13] Intel Corp. *Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2013. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[IZB11] Doug Inkster, Marcin Zukowski, and Peter Boncz. Integration of Vectorwise with Ingres. *ACM SIGMOD Record*, 40(3):45–53, 2011.

[JNS+97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proc. VLDB*, 1997.

[KBK02] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ACM SIGPLAN NOTICES*, pages 211–222. ACM, 2002.

[Khr13] Khronos Group. *OpenCL 2.0 - The open standard for parallel programming of heterogeneous systems*, July 2013. `http://www.nvidia.com/object/cuda_home_new.html`.

[KKG+11] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core CPUs. *Proceedings of the VLDB Endowment*, 5(1):61–72, 2011.

[Klu82] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM (JACM)*, 29(3):699–717, 1982.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[KS67] K Kahng and Simon M Sze. A floating gate and its application to memory devices. *Electron Devices, IEEE Transactions on*, 14(9):629–629, 1967.

[LB13] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 2013.

[LBM+07] Christian A. Lang, Bishwaranjan Bhattacharjee, Timothy Malkemus, Sriram Padmanabhan, and Kwai Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *ICDE*, Istanbul, Turkey, 2007.

[LCF+13] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L Price, Srikumar Rangarajan, Remus Rusanu, et al. Enhancements to SQL server column stores. In *Proceedings of the 2013 international conference on Management of data*, pages 1159–1168. ACM, 2013.

[LCH+11] Per-Åke Larson, Cipri Clinciu, Eric N Hanson, Artem Oks, Susan L Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. Sql server column store indexes. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1177–1184. ACM, 2011.

[Lev09] Levinthal, David. *Performance Analysis Guide for Intel® Core™ i7 Processors and Intel® Xeon 5500 Processors*, 2009.

[LFV+12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[LKF+13] Juchang Lee, Yong Sik Kwon, Franz Farber, Michael Muehle, Chulwon Lee, Christian Bensberg, Joo Yeon Lee, Arthur H Lee, and Wolfgang Lehner. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1165–1173. IEEE, 2013.

[Lus11] Ala Luszczak. Simple solutions for compressed execution in vectorized database system. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, July 2011.

[Man02] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, December 2002.

[Mar] Markus Oberhumer. *LZO: real-time data compression library*. `http://www.oberhumer.com/opensource/lzo/`.

[MC05] D. Metzler and W. Bruce Croft. A markov random field model for term dependencies. In *Proc. SIGIR*, 2005.

[MG12] Abhijeet Mohapatra and Michael Genesereth. Incrementally maintaining run-length encoded attributes in column stores. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, IDEAS '12, pages 146–154, New York, NY, USA, 2012. ACM.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1), 1992.

[Mic] Microsoft. *SQL Server*. `http://www.microsoft.com/sqlserver`.

[MIP] MIPS Technologies, Inc. *MIPS Architecture*. `https://www.mips.com/products/product-materials/processor/mips-architecture/`.

[MKD99] K. Mahesh, J. Kud, and P. Dixen. Oracle at TREC 8: a lexical approach. In *Proc. TREC*, 1999.

[MME13] R. Micheloni, A. Marelli, and K. Eshghi. *Inside Solid State Drives (SSDs)*. 2013.

[Mon] MonetDB. *MonetDB - An Open-Source Database System*. `http://www.monetdb.org`.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MSLdV14] Hannes Mühleisen, Thaer Samar, Jimmy Lin, and Arjen de Vries. Old dogs are great at new tricks: Column stores for ir prototyping. 2014.

[Nag10] Fabian Nagel. Recycling intermediate results in pipelined query evaluation. Master's thesis, Eberhard Karls Universität Tübingen, Tübingen, Germany, November 2010.

[NBV13] Fabian Nagel, Peter Boncz, and Stratis D Viglas. Recycling in pipelined query evaluation. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 338–349. IEEE, 2013.

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[OQ97] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *ACM Sigmod Record*, volume 26, pages 38–49. ACM, 1997.

[Par] Paraccel Analytic Database. `http://www.paraccel.com`.

[Pat04] David A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, October 2004.

[PBM+03] Sriram Padmanabhan, Bishwaranjan Bhattacharjee, Tim Malkemus, Leslie Cranston, and Matthew Huras. Multi-dimensional clustering: a new data layout scheme in db2. In *Proc. SIGMOD*, 2003.

[PGK88] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[PMAJ01] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proc. ICDE*, 2001.

[Por80] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[PP03] Meikel Pöss and Dmitry Potapov. Data Compression in Oracle. In *Proc. VLDB*, 2003.

[RAB+13] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. VLDB Endow.*, 6(11), August 2013.

[RBZ13] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 international conference on Management of data*, pages 1231–1242. ACM, 2013.

[RDS03] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A Case for Fractured Mirrors. *The VLDB Journal*, 12(2):89–101, 2003.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1), February 1992.

[Ros02] Kenneth A. Ross. Conjunctive selection conditions in main memory. In *Proc. PODS*, pages 109–120, 2002.

[RTK06] Thomas Rölleke, Theodora Tsikrika, and Gabriella Kazai. A general matrix framework for modelling information retrieval. *Information processing & management*, 42(1):4–30, 2006.

[RĬ2] Bogdan Răducanu. Micro adaptivity in a vectorized database system. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, August 2012.

[RvH93] Mark Roth and Scott van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, September 1993.

[RWB98] S. E. Robertson, S. Walker, and M. Beaulieu. Okapi at TREC-7: automatic ad hoc, filtering, VLC and interactive track. In *Proc. TREC*, 1998.

[Ś11] Michal Świtakowski. Integrating Cooperative Scans in a column-oriented DBMS. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, August 2011.

[SAC+79] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[SAN] SAND CDBMS. `http://www.sand.com`.

[ŚBZ12] Michał Świtakowski, Peter Boncz, and Marcin Zukowski. From cooperative scans to predictive buffer management. *Proceedings of the VLDB Endowment*, 5(12):1759–1770, 2012.

[SC05] Michael Stonebraker and Ugur Cetintemel. "one size fits all": an idea whose time has come and gone. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 2–11. IEEE, 2005.

[SE09] Dominik Slezak and Victoria Eastwood. Data warehouse technology by InfoBright. In *Proc. SIGMOD*, 2009.

[SFL+12] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 731–742. ACM, 2012.

[SKS01] Abraham Silberschatz, Henry F. Korth, and S. Sudershan. *Database System Concepts*. McGraw-Hill, Inc., July 2001.

[SL76] D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.*, 1(3), 1976.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *Proc. VLDB*, Vienna, Austria, 2007.

[Som11] Juliusz Sompolski. Just-in-time compilation in vectorized query execution. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, August 2011.

[SR13] Michael Stonebraker and Judy Robertson. Big Data is 'Buzzword du Jour;' CS Academics 'Have the Best Job'. *Communications of the ACM*, 56(9):10–11, 2013.

[Sta09] William Stallings. *Operating Systems: Internals and Design Principles, 6/E*. Prentice Hall, 2009.

[Sto05] M. Stonebraker et al. C-Store: A Column-oriented DBMS. In *Proc. VLDB*, 2005.

[Syb] Sybase Inc. *Sybase IQ*. `http://www.sybase.com/products/datawarehousing/sybaseiq`.

[SZB11] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40. ACM, 2011.

[Tat01] Simon Tatham. Counted B-trees. In *www.chiark.greenend.org.uk/sgtatham/algorithms/cbtree.html*, 2001.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, New York, NY, USA, 1995. ACM.

[Ter] Teradata Columnar. `www.teradata.com/database/Teradata-Columnar/`.

[Til13] Tilera Corp. *TILE-Gx Processor Family*, July 2013. `http://www.tilera.com/products/processors/TILE-Gx_Family`.

[Tra02] Transaction Processing Performance Council. *TPC Benchmark H version 2.1.0*, 2002. `http://www.tpc.org`.

[Tro03] Andrew Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.

[TWS04] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proc. VLDB*, pages 648–659, Toronto, Canada, 2004.

[UGA+09] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Predictable performance for unpredictable workloads. *Proceedings of the VLDB Endowment*, 2(1):706–717, 2009.

[Val87] Patrick Valduriez. Join indices. *ACM Transactions on Database Systems (TODS)*, 12(2):218–246, 1987.

[Ver] Vertica. `http://www.vertica.com`.

[Wel84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[Wil91] Ross N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference*, pages 362–371, 1991.

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, September 2000.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., 1999.

[YDS09] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web*, pages 401–410. ACM, 2009.

[ZBNH05] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. MonetDB/X100-A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

[ZHB06] Marcin Zukowski, Sándor Héman, and Peter Boncz. Architecture-conscious hashing. In *Proceedings of the 2nd international workshop on Data management on new hardware*, page 6. ACM, 2006.

[ZHNB06] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. ICDE*, 2006.

[ZHNB07] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *Proceedings of the 33rd international conference on Very large data bases*, pages 723–734. VLDB Endowment, 2007.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZLS08] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, pages 387–396. ACM, 2008.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38:2006, 2006.

[ZNB08] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th international workshop on Data management on new hardware*, pages 47–54. ACM, 2008.

[ZR02] Jingren Zhou and Kenneth A Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156. ACM, 2002.

[ZR03a] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proc. VLDB*, 2003.

[ZR03b] Jingren Zhou and Kenneth A. Ross. A multi-resolution block storage model for database design. In *Proc. IDEAS*, 2003.

[ZR04] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *Proc. SIGMOD*, 2004.

[Żuk09] Marcin Żukowski. *Balancing vectorized query execution with bandwidth-optimized storage.* Ph.d. thesis, Universiteit van Amsterdam, September 2009.

[ZvdWB12] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. Vectorwise: A vectorized analytical DBMS. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1349–1350. IEEE, 2012.

Summary

Updating compressed column-stores

Our modern society is relying more and more on information technology to manage both data and everyday activities. Over the years, this has resulted in an exponential growth in the amount of information being processed and stored by computer systems. Since the 1970's, *database management systems* (DBMS) have been used to organize, query and manipulate digital information in a structured way. Early database systems focused mainly on *transaction processing*, which deals with retrieving and manipulating data items, like, for example, the balance of a bank account, or address information of a customer. With time, the rapidly growing volume of data being gathered by organizations has sparkled a desire to gain more high-level insight into such data, for the purpose of providing, for example, historical or predictive views of business operation. This novel, *analytic* kind of data processing is often termed *business intelligence*, and is characterized by an "ad-hoc" query style, often involving large volumes of data, combined with a competitive or economic advantage if queries complete rapidly.

The growth in data volume has been accompanied by exponential improvements in certain areas of computer hardware, most notably CPU processing power and storage capacity of both main memory and disk. However, developments in data access *latency* and data transfer *bandwidth* have been lagging behind this exponential trend. The consequence being, that software developers have to deal with a widening gap between processing power on one hand, and data access cost on the other. Furthermore, the block-oriented access mechanisms provided by storage devices increasingly favor sequential over random access patterns, as latency improvements are falling behind advances in transfer bandwidth.

For analytic database workloads, which typically process large amounts of data, these trends have led to the wide adoption of so-called *column-oriented* storage layouts. Most database systems implement the relational model, where data is stored in two-dimensional *tables*, consisting of *rows* and *columns*. Each row, or *tuple* represents some entity, while each column represents an *attribute* of each entity. In a column-oriented, or *decomposed storage model* (DSM), tables are partitioned vertically on disk, meaning that attribute values from a single column are stored contiguously within fixed sized pages – the unit of transfer – on disk. The advantage being, that for queries requiring access to only a subset of table columns, it suffices to scan only the relevant attribute pages from disk, resulting in considerable I/O bandwidth savings compared to a *row-oriented*, or *N-ary storage model* (NSM), where a page always contains a sequence of complete tuples.

But column-oriented storage also provides benefits in terms of data processing efficiency within the CPU. By reading small fragments, or *vectors*, of attribute values, from a page in memory into the CPU cache, we make optimal use of cache-lines – the unit of transfer between memory and CPU – exposing comparable bandwidth savings as obtained by reading column-oriented pages

from disk. Furthermore, database *primitives*, responsible for performing actual computations on vectors of data, can be implemented as simple and efficient loops over arrays that are kept free of function calls and branching logic, enabling compilers to produce highly CPU-efficient code. Research around this *vectorized execution model* has resulted in the column-oriented *MonetDB/X100* DBMS, which was later commercialized under the name *Vectorwise*.

The work in this thesis provides solutions to two problems that may arise in a column-oriented and vectorized database engine like Vectorwise: (*i*) the danger of I/O bandwidth bottlenecks, which may occur even when using DSM, and (*ii*) how to provide efficient transactional update support on a storage layout that is notoriously unfriendly to in-place updates.

Given the growing gap between CPU power and disk transfer bandwidth, it may be impossible to deliver raw data from disk at a rate that matches the computational power of a vectorized engine. We show that data compression can be used to significantly alleviate such I/O bottlenecks, on both analytic and information retrieval workloads. The idea is to read compressed pages from disk, and decompress them at a minimal CPU cost, so that perceived I/O bandwidth can be improved, at most by a factor equal to the compression ratio. Here, we trade CPU processing power, which evolves favorably, for perceived disk bandwidth, which is scarce.

For such techniques to be effective even on high-performance disk systems, like RAID or SSD, it is important for (de)compression methods to be *lightweight*. We contribute three new compression schemes, PFOR, PFOR-DELTA and PDICT, that are specifically designed for the super-scalar capabilities of modern CPUs. These algorithms exploit the fact that attribute values within a column are from the same domain and of equal type, allowing (de)compression to be kept simple but fast (i.e. gigabytes per second). Furthermore, we show that by decompressing compressed pages in memory only at small granularity, in an on-demand fashion, directly *into the CPU cache*, we can eliminate the danger of memory bandwidth becoming a bottleneck.

To add transactional update support to a column-store DBMS, we consider in-place updates to be out of question, due to I/O costs that are proportional to the number of table attributes, and the fact that tables are often stored sorted and compressed. Rather, we focus on differential update techniques, where delta changes against otherwise immutable tables from a scan-optimized "read-store" are maintained in a memory-resident, update-friendly "write-store". During a scan, write-store updates are merged with the read-store table to produce an up-to-date table image. The idea is to sacrifice main-memory storage, which evolves favorably, to reduce I/O accesses, which are expensive, both in terms of latency and bandwidth.

We propose to organize delta updates against a read-store table by update *position*, or tuple offset, in a novel index structure called *positional delta tree* (PDT). Alternatively, one could organize update tuples by their (sort) key attribute *values*, for example in a B^+-tree-like structure. We show that positional update maintenance has several advantages over a value-based approach, most notably in the area of merging efficiency during a table scan. Furthermore, PDTs allow efficient encoding of attribute level modifications (i.e. SQL UPDATE), and can be stacked on top of each other, allowing for a clean implementation of snapshot isolation to support transactions.

Samenvatting

Wijzigingen aanbrengen in gecomprimeerde kolom-opslag

De hedendaagse maatschappij vertrouwt steeds vaker op informatietechnologie om zowel gegevens als dagelijkse activiteiten te beheren. Door de jaren heen heeft dit geleid tot een exponentiële groei in de hoeveelheid informatie die wordt verwerkt en opgeslagen door computers. Sinds 1970 zijn *database management systemen* (DBMS) gebruikt om digitale informatie op gestructureerde wijze te organiseren, uit te vragen en te manipuleren. Vroege databasesystemen richtten zich hoofdzakelijk op *transactie verwerking*, dat zich bezigt met het ophalen en manipuleren van data items, zoals, bijvoorbeeld, het saldo op een bankrekening of adresgegevens van een klant. Met de tijd heeft het snel groeiende volume van door organisaties verzamelde gegevens een behoefte aangewakkerd om een meer globaal inzicht te krijgen in dergelijke gegevens, met als doel het verstrekken van, bijvoorbeeld, historische of voorspellende inzichten aangaande de bedrijfsvoering. Deze nieuwe, *analytische* manier van data verwerken wordt vaak *business intelligence* genoemd, en wordt gekarakteriseerd door een "ad-hoc" query (vraag) patroon, vaak over grote hoeveelheden data, in combinatie met een competitief of economisch voordeel indien deze vragen snel beantwoord worden.

De groei in datavolume is gepaard gegaan met een exponentiële vooruitgang binnen bepaalde eigenschappen van computerhardware, met name verwerkingskracht van processoren (CPUs) en opslagcapaciteit van zowel werkgeheugen als schijfopslag. Echter, zowel de *responstijd* ten aanzien van data verzoeken als de *bandbreedte* voor het transport van de data zijn bij deze exponentiële trend achtergebleven. Het gevolg is dat softwareontwikkelaars geconfronteerd worden met een groeiende discrepantie tussen verwerkingskracht aan de ene kant, en trage toegang tot data aan de andere. Voorts bevoordelen de blok-geörienteerde toegangsmethoden, die opslag-electonica verstrekken, in toenemende mate sequentiële boven lukrake toegangspatronen, aangezien ontwikkelingen in responstijd achterblijven bij ontwikkelingen rond bandbreedte.

Voor analytische database toepassingen, die over het algemeen grote hoeveelheden data verwerken, hebben deze trends geleid tot brede acceptatie van zogenaamde *kolom-georiënteerde* opslagtechnieken. De meeste databasesystemen hanteren een relationeel model, waarbinnen gegevens worden opgeslagen in tweedimensionale *tabellen*, bestaande uit *rijen* en *kolommen*. Elke rij, of *tuple*, vertegenwoordigt een zekere entiteit, waarbij elke kolom een *attribuut*, of eigenschap, van die entiteit voorstelt. In een kolom-geörienteerd opslagmodel, het zogeheten *decomposed storage model* (DSM), worden tabellen verticaal gepartitioneerd opgeslagen, wat inhoudt dat attribuutwaarden uit een enkele kolom aaneengesloten worden opgeslagen binnen een *pagina* – de transporteenheid tussen schijf en geheugen – van vaste grootte. Het voordeel is dat voor vragen die slechts toegang tot een deelverzameling van kolommen in een tabel nodig hebben, het volstaat om alleen de pagina's van relevante kolommen van schijf te lezen, wat kan leiden tot een substantiële besparing van schijfbandbreedte ten opzichte van een rij-georiënteerd opslagmodel, het zogeheten *N-ary storage model* (NSM), waar een pagina altijd een reeks volledige tuples bevat.

Maar kolom-georiënteerde opslag biedt ook voordelen op het gebied van dataverwerkingsefficiëntie binnen de processor. Door kleine fragmenten, of *vectoren*, met attribuutwaarden uit een pagina binnen het geheugen direct naar de CPU-cache te lezen, wordt optimaal gebruik gemaakt van cachelijnen (de transporteenheid tussen geheugen en CPU), waarmee vergelijkbare bandbreedtebesparingen kunnen worden behaald als bij het lezen van kolom-georiënteerde pagina's van disk. Tevens kunnen database *primitieven*, verantwoordelijk voor het uitvoeren van de uiteindelijke berekeningen over vectoren met data, worden geïmplementeerd als simpele en efficiënte lussen (loops) over reeksen (arrays), die vrij zijn van functieaanroepen en vertakkingen, wat compilers in staat stelt om CPU-efficiënte code te genereren. Onderzoek omtrent dit *gevectoriseerde uitvoerings model* heeft geresulteerd in het kolom-georiënteerde *MonetDB/X100* DBMS, dat later is gecommercialiseerd onder de noemer *Vectorwise*.

Het werk in deze dissertatie biedt oplossingen voor twee problemen die kunnen voorkomen bij kolom-georiënteerde en gevectoriseerde databasesystemen zoals Vectorwise: *(i)* een gevaar voor knelpunten aangaande input/output (I/O) bandbreedte, die zelfs bij gebruik van DSM nog kunnen voorkomen, en *(ii)* hoe te voorzien in functionaliteit die het mogelijk maakt om op efficiënte wijze transactionele wijzigingen aan te brengen binnen een opslagstructuur die uiterst ongeschikt is voor het aanbrengen van lokale wijzigingen.

Gegeven de groeiende discrepantie tussen processorkracht enerzijds en schijfbandbreedte anderzijds, is het soms onmogelijk om ruwe data vanaf schijf aan te leveren met een snelheid die overeenkomt met de verwerkingskracht van een gevectoriseerde databasekern. We tonen aan dat datacompressie gebruikt kan worden om dergelijke knelpunten significant te verlichten, zowel bij analytische als IR (information retrieval) toepassingen. Het idee is om gecomprimeerde pagina's van schijf te lezen, en onder minimale CPU-belasting te decomprimeren, zodat de waargenomen I/O-bandbreedte verbeterd kan worden, met maximaal een factor gelijk aan de compressie ratio. We verruilen hier CPU-rekenkracht, hetgeen zich gunstig ontwikkelt, tegen waargenomen schijfbandbreedte, hetgeen schaars is.

Om dergelijke technieken ook op snelle schijfsystemen, zoals RAID (redundant array of independent disks) of SSD (solid-state drive), te laten werken, is het van belang dat (de)compressiemethoden *lichtgewicht* zijn. We dragen drie nieuwe compressie schema's bij, PFOR, PFOR-DELTA en PDICT, die specifiek zijn ontworpen voor de superscalaire infrastructuur van moderne CPUs. Deze algoritmen maken gebruik van de observatie dat waarden binnen een kolom allen uit hetzelfde domein afkomstig zijn en hetzelfde datatype hebben, wat ons in staat stelt om (de)compressie simpel en snel – gigabytes per seconde – te houden. Tevens tonen we aan dat door gecomprimeerde pagina's binnen het werkgeheugen slechts met lage granulariteit, naar behoefte, te decomprimeren, direct *naar de CPU-cache*, we kunnen voorkomen dat geheugenbandbreedte een knelpunt wordt.

Om transactionele wijzigingen aan te brengen binnen een kolom-opslag DBMS, beschouwen we het lokaal aanbrengen van wijzigingen als niet afdoende, vanwege het gegeven dat de I/O kosten proportioneel stijgen met het aantal attributen van een tabel. Liever richten wij ons tot differentiële wijzigingstechnieken, waar wijzigingen ten opzichte van anderszins onveranderlijke tabellen in een voor scans geoptimaliseerde "lees-opslag", worden bijgehouden in een makkelijk te wijzigen "schrijf-opslag" in het werkgeheugen. Tijdens een scan worden wijzig-

ingen uit de schrijf-opslag vervlochten met de tabel uit lees-opslag om de huidige toestand van de tabel te reproduceren. Het idee is om werkgeheugencapaciteit, hetgeen zich gunstig ontwikkelt, te verruilen voor een reductie in het aantal I/O operaties, die kostbaar zijn, zowel in termen van responstijd als bandbreedte.

We stellen voor om differentiële wijzigingen ten opzichte van een tabel in lees-opslag bij te houden op basis van tuple *positie*, in een nieuwe boomvormige indexstructuur genaamd *positional delta tree* (PDT). Alternatief zou men wijzigingen kunnen bijhouden en rangschikken op basis van de *waarde* van (sorteer) sleutelattributen. We tonen aan dat het positioneel bijhouden van wijzigingen voordelen biedt ten opzichte van een op waarden gebaseerde aanpak, met name bij het vervlechten van wijzigingen tijdens een scan. Tevens maken PDTs een efficiënte encodering van modificatie (SQL UPDATE) mogelijk, en kunnen zij boven op elkaar gestapeld worden, wat een overzichtelijke implementatie van momentopname isolatie (snapshot isolation) levert om te voorzien in transacties.

SIKS Dissertatiereeks

2009-01 Rasa Jurgelenaite (RUN) Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU) Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT) A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU) Understanding Classification

2009-07 Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UVA) Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UVA) Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) Operating Guidelines for Services

2009-13 Steven de Jong (UM) Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)

2009-15 Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT) New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT) Feature Extraction from Visual Data

2009-18 Fabian Groffen (CWI) Armada, An Evolving Database System

2009-19 Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 Bob van der Vecht (UU) Adjustable Autonomy: Controling Influences on Decision Making

2009-21 Stijn Vanderlooy (UM) Ranking and Reliable Classification

2009-22 Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence

2009-23 Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment

2009-24 Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations

2009-25 Alex van Ballegooij (CWI) "RAM: Array Database Management through Relational Mapping"

2009-26 Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 Christian Glahn (OU) Contextual Support of social Engagement and Reflection on the Web

2009-28 Sander Evers (UT) Sensor Data Management with Probabilistic Models

2009-29 Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage

2009-31 Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors

2009-33 Khiet Truong (UT) How Does Real Affect Affect Affect Recognition In Speech?

2009-34 Inge van de Weerd (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 Wouter Koelewijn (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 Marco Kalz (OUN) Placement Support for Learners in Learning Networks

2009-37 Hendrik Drachsler (OUN) Navigation Support for Learners in Informal Learning Networks

2009-38 Riina Vuorikari (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin) Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 Stephan Raaijmakers (UvT) Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 Igor Berezhnyy (UvT) Digital Analysis of Paintings

2009-42 Toine Bogers (UvT) Recommender Systems for Social Bookmarking

2009-43 Virginia Nunes Leal Franqueira (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 Roberto Santana Tapia (UT) Assessing Business-IT Alignment in Networked Organizations

2009-45 Jilles Vreeken (UU) Making Pattern Mining Useful

2009-46 Loredana Afanasiev (UvA) Querying XML: Benchmarks and Recursion

2010-01 Matthijs van Leeuwen (UU) Patterns that Matter

2010-02 Ingo Wassink (UT) Work flows in Life Science

2010-03 Joost Geurts (CWI) A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 Olga Kulyk (UT) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 Claudia Hauff (UT) Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 Sander Bakkes (UvT) Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT) Gesture interaction at a Distance

2010-08 Krzysztof Siewicz (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 Hugo Kielman (UL) A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging

2010-10 Rebecca Ong (UL) Mobile Communication and Protection of Children

2010-11 Adriaan Ter Mors (TUD) The world according to MARP: Multi-Agent Route Planning

2010-12 Susan van den Braak (UU) Sensemaking software for crime analysis

2010-13 Gianluigi Folino (RUN) High Performance Data Mining using Bio-inspired techniques

2010-14 Sander van Splunter (VU) Automated Web Service Reconfiguration

2010-15 Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models

2010-16 Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice

2010-17 Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

2010-18 Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation

2010-19 Henriette Cramer (UvA) People's Responses to Autonomous and Adaptive Systems

2010-20 Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative

2010-21 Harold van Heerde (UT) Privacy-aware data management by means of data degradation

2010-22 Michiel Hildebrand (CWI) End-user Support for Access to
Heterogeneous Linked Data

2010-23 Bas Steunebrink (UU) The Logical Structure of Emotions

2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies

2010-25 Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective

2010-26 Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines

2010-27 Marten Voulon (UL) Automatisch contracteren

2010-28 Arne Koopman (UU) Characteristic Relational Patterns

2010-29 Stratos Idreos(CWI) Database Cracking: Towards Auto-tuning Database Kernels

2010-30 Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval

2010-31 Victor de Boer (UVA) Ontology Enrichment from Heterogeneous Sources on the Web

2010-32 Marcel Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems

2010-33 Robin Aly (UT) Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval

2010-34 Teduh Dirgahayu (UT) Interaction Design in Service Compositions

2010-35 Dolf Trieschnigg (UT) Proof of Concept: Concept-based Biomedical Information Retrieval

2010-36 Jose Janssen (OU) Paving the Way for

Lifelong Learning; Facilitating competence development through a learning path specification

2010-37 Niels Lohmann (TUE) Correctness of services and their composition

2010-38 Dirk Fahland (TUE) From Scenarios to components

2010-39 Ghazanfar Farooq Siddiqui (VU) Integrative modeling of emotions in virtual agents

2010-40 Mark van Assem (VU) Converting and Integrating Vocabularies for the Semantic Web

2010-41 Guillaume Chaslot (UM) Monte-Carlo Tree Search

2010-42 Sybren de Kinderen (VU) Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach

2010-43 Peter van Kranenburg (UU) A Computational Approach to Content-Based Retrieval of Folk Song Melodies

2010-44 Pieter Bellekens (TUE) An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain

2010-45 Vasilios Andrikopoulos (UvT) A theory and model for the evolution of software services

2010-46 Vincent Pijpers (VU) e3alignment: Exploring Inter-Organizational Business-ICT Alignment

2010-47 Chen Li (UT) Mining Process Model Variants: Challenges, Techniques, Examples

2010-48 Withdrawn

2010-49 Jahn-Takeshi Saito (UM) Solving difficult game positions

2010-50 Bouke Huurnink (UVA) Search in Audiovisual Broadcast Archives

2010-51 Alia Khairia Amin (CWI) Understanding and supporting information seeking tasks in multiple sources

2010-52 Peter-Paul van Maanen (VU) Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention

2010-53 Edgar Meij (UVA) Combining Concepts and Language Models for Information Access

2011-01 Botond Cseke (RUN) Variational Algorithms for Bayesian Inference in Latent Gaussian Models

2011-02 Nick Tinnemeier(UU) Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language

2011-03 Jan Martijn van der Werf (TUE) Compositional Design and Verification of Component-Based Information Systems

2011-04 Hado van Hasselt (UU) Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference

2011-05 Base van der Raadt (VU) Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.

2011-06 Yiwen Wang (TUE) Semantically-Enhanced Recommendations in Cultural Heritage

2011-07 Yujia Cao (UT) Multimodal Information Presentation for High Load Human Computer Interaction

2011-08 Nieske Vergunst (UU) BDI-based Generation of Robust Task-Oriented Dialogues

2011-09 Tim de Jong (OU) Contextualised Mobile Media for Learning

2011-10 Bart Bogaert (UvT) Cloud Content Contention

2011-11 Dhaval Vyas (UT) Designing for Awareness: An Experience-focused HCI Perspective

2011-12 Carmen Bratosin (TUE) Grid Architecture for Distributed Process Mining

2011-13 Xiaoyu Mao (UvT) Airport under Control. Multiagent Scheduling for Airport Ground Handling

2011-14 Milan Lovric (EUR) Behavioral Finance and Agent-Based Artificial Markets

2011-15 Marijn Koolen (UvA) The Meaning of Structure: the Value of Link Evidence for Information Retrieval

2011-16 Maarten Schadd (UM) Selective Search in Games of Different Complexity

2011-17 Jiyin He (UVA) Exploring Topic Structure: Coherence, Diversity and Relatedness

2011-18 Mark Ponsen (UM) Strategic Decision-Making in complex games

2011-19 Ellen Rusman (OU) The Mind ' s Eye on Personal Profiles

2011-20 Qing Gu (VU) Guiding service-oriented software engineering - A view-based approach

2011-21 Linda Terlouw (TUD) Modularization and Specification of Service-Oriented Systems

2011-22 Junte Zhang (UVA) System Evaluation of Archival Description and Access

2011-23 Wouter Weerkamp (UVA) Finding People and their Utterances in Social Media

2011-24 Herwin van Welbergen (UT) Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior

2011-25 Syed Waqar ul Qounain Jaffry (VU)) Analysis and Validation of Models for Trust Dynamics

2011-26 Matthijs Aart Pontier (VU) Virtual Agents for Human Communication - Emotion

Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots

2011-27 Aniel Bhulai (VU) Dynamic website optimization through autonomous management of design patterns

2011-28 Rianne Kaptein(UVA) Effective Focused Retrieval by Exploiting Query Context and Document Structure

2011-29 Faisal Kamiran (TUE) Discrimination-aware Classification

2011-30 Egon van den Broek (UT) Affective Signal Processing (ASP): Unraveling the mystery of emotions

2011-31 Ludo Waltman (EUR) Computational and Game-Theoretic Approaches for Modeling Bounded Rationality

2011-32 Nees-Jan van Eck (EUR) Methodological Advances in Bibliometric Mapping of Science

2011-33 Tom van der Weide (UU) Arguing to Motivate Decisions

2011-34 Paolo Turrini (UU) Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations

2011-35 Maaike Harbers (UU) Explaining Agent Behavior in Virtual Training

2011-36 Erik van der Spek (UU) Experiments in serious game design: a cognitive approach

2011-37 Adriana Burlutiu (RUN) Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference

2011-38 Nyree Lemmens (UM) Bee-inspired Distributed Optimization

2011-39 Joost Westra (UU) Organizing Adaptation using Agents in Serious Games

2011-40 Viktor Clerc (VU) Architectural Knowledge Management in Global Software Development

2011-41 Luan Ibraimi (UT) Cryptographically Enforced Distributed Data Access Control

2011-42 Michal Sindlar (UU) Explaining Behavior through Mental State Attribution

2011-43 Henk van der Schuur (UU) Process Improvement through Software Operation Knowledge

2011-44 Boris Reuderink (UT) Robust Brain-Computer Interfaces

2011-45 Herman Stehouwer (UvT) Statistical Language Models for Alternative Sequence Selection

2011-46 Beibei Hu (TUD) Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work

2011-47 Azizi Bin Ab Aziz(VU) Exploring Computational Models for Intelligent Support of Persons with Depression

2011-48 Mark Ter Maat (UT) Response Selection and Turn-taking for a Sensitive Artificial Listening Agent

2011-49 Andreea Niculescu (UT) Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012-01 Terry Kakeeto (UvT) Relationship Marketing for SMEs in Uganda

2012-02 Muhammad Umair(VU) Adaptivity, emotion, and Rationality in Human and Ambient Agent Models

2012-03 Adam Vanya (VU) Supporting Architecture Evolution by Mining Software Repositories

2012-04 Jurriaan Souer (UU) Development of Content Management System-based Web Applications

2012-05 Marijn Plomp (UU) Maturing Interorganisational Information Systems

2012-06 Wolfgang Reinhardt (OU) Awareness Support for Knowledge Workers in Research Networks

2012-07 Rianne van Lambalgen (VU) When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions

2012-08 Gerben de Vries (UVA) Kernel Methods for Vessel Trajectories

2012-09 Ricardo Neisse (UT) Trust and Privacy Management Support for Context-Aware Service Platforms

2012-10 David Smits (TUE) Towards a Generic Distributed Adaptive Hypermedia Environment

2012-11 J.C.B. Rantham Prabhakara (TUE) Process Mining in the Large: Preprocessing, Discovery, and Diagnostics

2012-12 Kees van der Sluijs (TUE) Model Driven Design and Data Integration in Semantic Web Information Systems

2012-13 Suleman Shahid (UvT) Fun and Face: Exploring non-verbal expressions of emotion during playful interactions

2012-16 Fiemke Both (VU) Helping people by understanding them - Ambient Agents supporting task execution and depression treatment

2012-17 Amal Elgammal (UvT) Towards a Comprehensive Framework for Business Process Compliance

2012-18 Eltjo Poort (VU) Improving Solution Architecting Practices

2012-20 Ali Bahramisharif (RUN) Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing

2012-22 Thijs Vis (UvT) Intelligence, politie en veiligheidsdienst: verenigbare grootheden?

2012-23 Christian Muehl (UT) Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction

2012-24 Laurens van der Werff (UT) Evaluation of Noisy Transcripts for Spoken Document Retrieval

2012-25 Silja Eckartz (UT) Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application

2012-26 Emile de Maat (UVA) Making Sense of Legal Text

2012-27 Hayrettin Gurkok (UT) Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games

2012-28 Nancy Pascall (UvT) Engendering Technology Empowering Women

2012-29 Almer Tigelaar (UT) Peer-to-Peer Information Retrieval

2012-30 Alina Pommeranz (TUD) Designing Human-Centered Systems for Reflective Decision Making

2012-31 Emily Bagarukayo (RUN) A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure

2012-32 Wietske Visser (TUD) Qualitative multi-criteria preference representation and reasoning

2012-33 Rory Sie (OUN) Coalitions in Cooperation Networks (COCOON)

2012-34 Pavol Jancura (RUN) Evolutionary analysis in PPI networks and applications

2012-35 Evert Haasdijk (VU) Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics

2012-36 Denis Ssebugwawo (RUN) Analysis and Evaluation of Collaborative Modeling Processes

2012-37 Agnes Nakakawa (RUN) A Collaboration Process for Enterprise Architecture Creation

2012-38 Selmar Smit (VU) Parameter Tuning and Scientific Testing in Evolutionary Algorithms

2012-39 Hassan Fatemi (UT) Risk-aware design of value and coordination networks

2012-40 Agus Gunawan (UvT) Information Access for SMEs in Indonesia

2012-41 Sebastian Kelle (OU) Game Design Patterns for Learning

2012-42 Dominique Verpoorten (OU) Reflection Amplifiers in self-regulated Learning

2012-43 Withdrawn

2012-44 Anna Tordai (VU) On Combining Alignment Techniques

2012-45 Benedikt Kratz (UvT) A Model and Language for Business-aware Transactions

2012-46 Simon Carter (UVA) Exploration and Exploitation of Multilingual Data for Statistical Machine Translation

2012-47 Manos Tsagkias (UVA) Mining Social Media: Tracking Content and Predicting Behavior

2012-48 Jorn Bakker (TUE) Handling Abrupt Changes in Evolving Time-series Data

2012-50 Steven van Kervel (TUD) Ontologogy driven Enterprise Information Systems Engineering

2012-51 Jeroen de Jong (TUD) Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

2013-01 Viorel Milea (EUR) News Analytics for Financial Decision Support

2013-02 Erietta Liarou (CWI) MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing

2013-03 Szymon Klarman (VU) Reasoning with Contexts in Description Logics

2013-04 Chetan Yadati(TUD) Coordinating autonomous planning and scheduling

2013-05 Dulce Pumareja (UT) Groupware Requirements Evolutions Patterns

2013-06 Romulo Goncalves(CWI) The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience

2013-07 Giel van Lankveld (UvT) Quantifying Individual Player Differences

2013-08 Robbert-Jan Merk(VU) Making enemies: cognitive modeling for opponent agents in fighter pilot simulators

2013-09 Fabio Gori (RUN) Metagenomic Data Analysis: Computational Methods and Applications

2013-10 Jeewanie Jayasinghe Arachchige(UvT) A Unified Modeling Framework for Service Design.

2013-11 Evangelos Pournaras(TUD) Multi-level Reconfigurable Self-organization in Overlay Services

2013-12 Marian Razavian(VU) Knowledge-driven Migration to Services

2013-13 Mohammad Safiri(UT) Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly

2013-14 Jafar Tanha (UVA) Ensemble Approaches to Semi-Supervised Learning Learning

2013-15 Daniel Hennes (UM) Multiagent Learning - Dynamic Games and Applications

2013-16 Eric Kok (UU) Exploring the practical benefits of argumentation in multi-agent deliberation

2013-17 Koen Kok (VU) The PowerMatcher: Smart Coordination for the Smart Electricity Grid

2013-18 Jeroen Janssens (UvT) Outlier Selection and One-Class Classification

2013-19 Renze Steenhuizen (TUD) Coordinated Multi-Agent Planning and Scheduling

2013-20 Katja Hofmann (UvA) Fast and Reliable Online Learning to Rank for Information Retrieval

2013-21 Sander Wubben (UvT) Text-to-text generation by monolingual machine translation

2013-22 Tom Claassen (RUN) Causal Discovery and Logic

2013-23 Patricio de Alencar Silva(UvT) Value Activity Monitoring

2013-24 Haitham Bou Ammar (UM) Automated Transfer in Reinforcement Learning

2013-25 Agnieszka Anna Latoszek-Berendsen (UM) Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System

2013-26 Alireza Zarghami (UT) Architectural Support for Dynamic Homecare Service Provisioning

2013-27 Mohammad Huq (UT) Inference-based Framework Managing Data Provenance

2013-28 Frans van der Sluis (UT) When Complexity becomes Interesting: An Inquiry into the Information eXperience

2013-29 Iwan de Kok (UT) Listening Heads

2013-30 Joyce Nakatumba (TUE) Resource-Aware Business Process Management: Analysis and Support

2013-31 Dinh Khoa Nguyen (UvT) Blueprint Model and Language for Engineering Cloud Applications

2013-32 Kamakshi Rajagopal (OUN) Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development

2013-33 Qi Gao (TUD) User Modeling and Personalization in the Microblogging Sphere

2013-34 Kien Tjin-Kam-Jet (UT) Distributed Deep Web Search

2013-35 Abdallah El Ali (UvA) Minimal Mobile Human Computer Interaction

2013-36 Than Lam Hoang (TUe) Pattern Mining in Data Streams

2013-37 Dirk Börner (OUN) Ambient Learning Displays

2013-38 Eelco den Heijer (VU) Autonomous Evolutionary Art

2013-39 Joop de Jong (TUD) A Method for Enterprise Ontology based Design of Enterprise Information Systems

2013-40 Pim Nijssen (UM) Monte-Carlo Tree Search for Multi-Player Games

2013-41 Jochem Liem (UVA) Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning

2013-42 Léon Planken (TUD) Algorithms for Simple Temporal Reasoning

2013-43 Marc Bron (UVA) Exploration and Contextualization through Interaction and Concepts

2014-01 Nicola Barile (UU) Studies in Learning Monotone Models from Data

2014-02 Fiona Tuliyano (RUN) Combining System Dynamics with a Domain Modeling Method

2014-03 Sergio Raul Duarte Torres (UT) Information Retrieval for Children: Search Behavior and Solutions

2014-04 Hanna Jochmann-Mannak (UT) Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation

2014-05 Jurriaan van Reijsen (UU) Knowledge Perspectives on Advancing Dynamic Capability

2014-06 Damian Tamburri (VU) Supporting Networked Software Development

2014-07 Arya Adriansyah (TUE) Aligning Observed and Modeled Behavior

2014-08 Samur Araujo (TUD) Data Integration over Distributed and Heterogeneous Data Endpoints

2014-09 Philip Jackson (UvT) Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language

2014-10 Ivan Salvador Razo Zapata (VU) Service Value Networks

2014-11 Janneke van der Zwaan (TUD) An Empathic Virtual Buddy for Social Support

2014-12 Willem van Willigen (VU) Look Ma, No Hands: Aspects of Autonomous Vehicle Control

2014-13 Arlette van Wissen (VU) Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains

2014-14 Yangyang Shi (TUD) Language Models With Meta-information

2014-15 Natalya Mogles (VU) Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare

2014-16 Krystyna Milian (VU) Supporting trial recruitment and design by automatically interpreting eligibility criteria

2014-17 Kathrin Dentler (VU) Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability

2014-18 Mattijs Ghijsen (VU) Methods and Models for the Design and Study of Dynamic Agent Organizations

2014-19 Vinicius Ramos (TUE) Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support

2014-20 Mena Habib (UT) Named Entity Extraction and Disambiguation for Informal Text: The Missing Link

2014-21 Kassidy Clark (TUD) Negotiation and Monitoring in Open Environments

2014-22 Marieke Peeters (UU) Personalized Educational Games - Developing agent-supported scenario-based training

2014-23 Eleftherios Sidirourgos (UvA/CWI) Space Efficient Indexes for the Big Data Era

2014-24 Davide Ceolin (VU) Trusting Semi-structured Web Data

2014-25 Martijn Lappenschaar (RUN) New network models for the analysis of disease interaction

2014-26 Tim Baarslag (TUD) What to Bid and When to Stop

2014-27 Rui Jorge Almeida (EUR) Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty

2014-28 Anna Chmielowiec (VU) Decentralized k-Clique Matching

2014-29 Jaap Kabbedijk (UU) Variability in Multi-Tenant Enterprise Software

2014-30 Peter de Cock (UvT) Anticipating Criminal Behaviour

2014-31 Leo van Moergestel (UU) Agent Technology in Agile Multiparallel Manufacturing and Product Support

2014-32 Naser Ayat (UvA) On Entity Resolution in Probabilistic Data

2014-33 Tesfa Tegegne (RUN) Service Discovery in eHealth

2014-34 Christina Manteli(VU) The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.

2014-35 Joost van Ooijen (UU) Cognitive Agents in Virtual Worlds: A Middleware Design Approach

2014-36 Joos Buijs (TUE) Flexible Evolutionary Algorithms for Mining Structured Process Models

2014-37 Maral Dadvar (UT) Experts and Machines United Against Cyberbullying

2014-38 Danny Plass-Oude Bos (UT) Making brain-computer interfaces better: improving usability through post-processing.

2014-39 Jasmina Maric (UvT) Web Communities, Immigration, and Social Capital

2014-40 Walter Omona (RUN) A Framework for Knowledge Management Using ICT in Higher Education

2014-41 Frederic Hogenboom (EUR) Automated Detection of Financial Events in News Text

2014-42 Carsten Eijckhof (CWI/TUD) Contextual Multidimensional Relevance Models

2014-43 Kevin Vlaanderen (UU) Supporting Process Improvement using Method Increments

2014-44 Paulien Meesters (UvT) Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.

2014-45 Birgit Schmitz (OUN) Mobile Games for Learning: A Pattern-Based Approach

2014-46 Ke Tao (TUD) Social Web Data Analytics: Relevance, Redundancy, Diversity

2014-47 Shangsong Liang (UVA) Fusion and Diversification in Information Retrieval

2015-01 Niels Netten (UvA) Machine Learning for Relevance of Information in Crisis Response

2015-02 Faiza Bukhsh (UvT) Smart auditing: Innovative Compliance Checking in Customs Controls

2015-03 Twan van Laarhoven (RUN) Machine learning for network data

2015-04 Howard Spoelstra (OUN) Collaborations in Open Learning Environments

2015-05 Christoph Bösch(UT) Cryptographically Enforced Search Pattern Hiding

2015-06 Farideh Heidari (TUD) Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes

2015-07 Maria-Hendrike Peetz(UvA) Time-Aware Online Reputation Analysis

2015-08 Jie Jiang (TUD) Organizational Compliance: An agent-based model for designing and evaluating organizational interactions

2015-09 Randy Klaassen(UT) HCI Perspectives on Behavior Change Support Systems

2015-10 Henry Hermans (OUN) OpenU: design of an integrated system to support lifelong learning

2015-11 Yongming Luo(TUE) Designing algorithms for big graph datasets: A study of computing bisimulation and joins

2015-12 Julie M. Birkholz (VU) Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks

2015-13 Giuseppe Procaccianti(VU) Energy-Efficient Software

2015-14 Bart van Straalen (UT) A cognitive approach to modeling bad news conversations

2015-15 Klaas Andries de Graaf (VU) Ontology-based Software Architecture Documentation

2015-16 Changyun Wei (UT) Cognitive Coordination for Cooperative Multi-Robot Teamwork

2015-17 André van Cleeff (UT) Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs

2015-18 Holger Pirk (CWI) Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories

2015-19 Bernardo Tabuenca (OUN) Ubiquitous Technology for Lifelong Learners

2015-20 Loïs Vanhée(UU) Using Culture and Values to Support Flexible Coordination

2015-21 Sibren Fetter (OUN) Using Peer-Support to Expand and Stabilize Online Learning

2015-23 Luit Gazendam (VU) Cataloguer Support in Cultural Heritage

2015-24 Richard Berendsen (UVA) Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation

2015-25 Steven Woudenberg (UU) Bayesian Tools for Early Disease Detection

2015-26 Alexander Hogenboom (EUR) Sentiment Analysis of Text Guided by Semantics and Structure